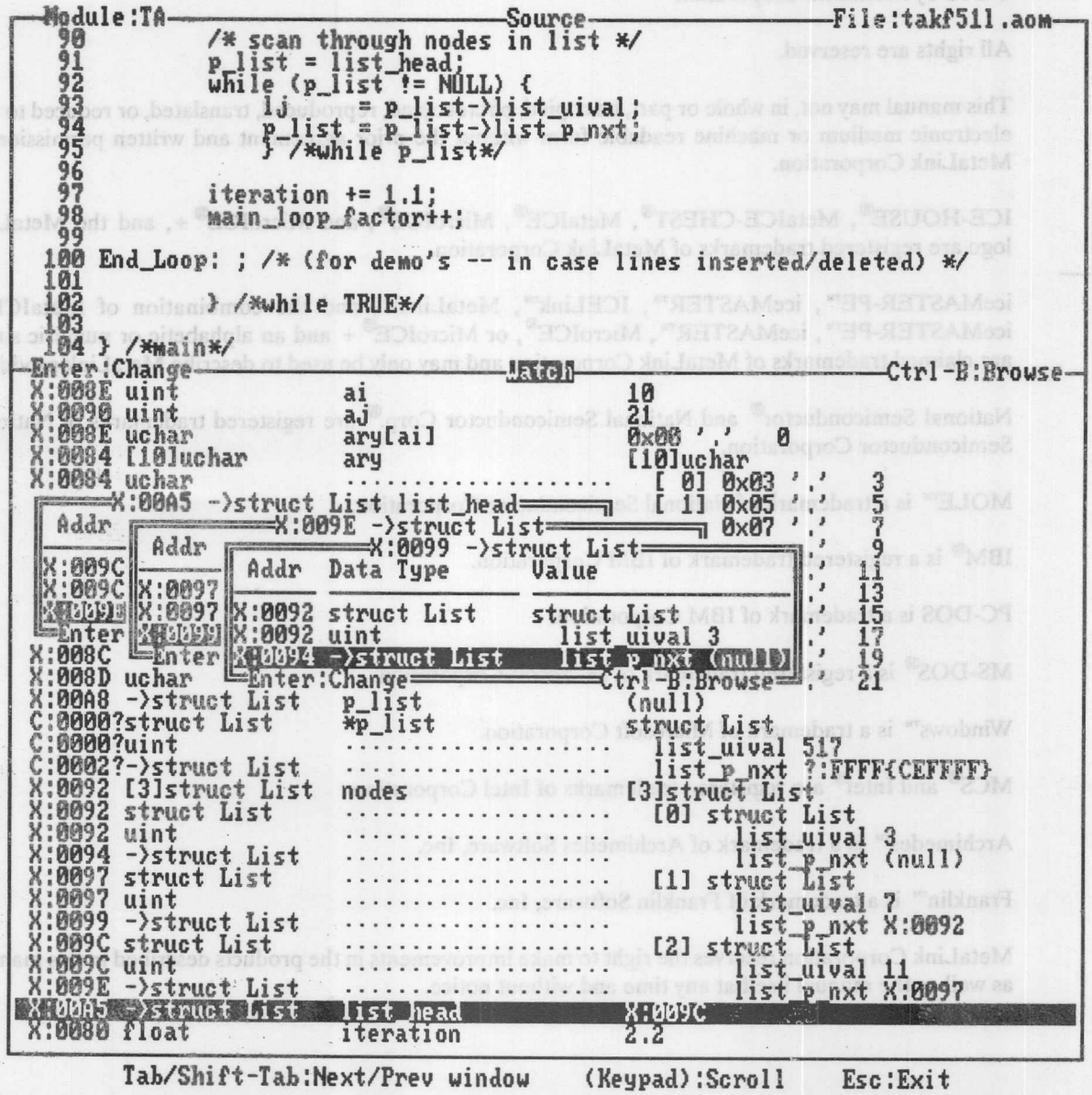
MetaLink® Corporation

iceMASTER™ Software Version 3.3 Supplement



Document Version: 1.1

Table of Contents

Chapter 1: Introduction

iceMASTER Host Software Version 3.3 contains these major enhancements:

- All address, length or count specifications can now be general expressions, rather than just a single symbol or constant.
- Support for long integer, single-precision (**float**) and double-precision (**double**) floating-point variables
  - Display
  - Input
- Full support for structures, unions, arrays and pointers
  - Display/Change
    * Watch Window
    * *Display/Alter|Expression* command
    * *Source/Symbols|Global* command
    * *Source/Symbols|Local* command
    * *Source/Symbols|Address* command
    * *Source/Symbols|Alpha* command
  - Full structure, union, array & pointer support for these compilers:
    * Keil/Franklin C51
    * IAR/Archimedes C-51
    * (others coming soon)
- Data Structure Browser/Inspector
  - Inspect structures, unions and arrays
  - Follow pointers
  - Change any value at any point along the way
  - Backtrack
- True Expressions in Watch Window (not just static addresses)
  - Indirection through NULL or invalid pointers is flagged in the display.
- OS-Escape ("Shell-Out") to DOS during Emulation
  - OS-Escape Hot Key: **Alt-O**
  - Emulation continues while working in DOS.
  - All DOS memory, except about 7K-10K bytes, available during all OS-Escapes
- Supports all *iceMASTER* hardware products
  - *iceMASTER* Model 200/400 Emulators (MCS-51, COP8, 68HC11 & 68HC05)
  - *iceMASTER*-PE Emulators
  - *iceMASTER*-COP8 Debug Modules

The following features were added in earlier versions of the *iceMASTER* host software. Depending on which *iceMASTER* hardware product you purchased, these features may or may not be documented in the primary manual:

- Source Window & Assembler/Disassembler Enhancements
  (software version 3.2 – see Appendix A)
  - "Source-Only" display mode option in Source Window
  - Assembler/Disassembler Window fully-scrollable
  - Set permanent break-points directly in Source or Assembler/Disassembler Windows
  - Set temporary break-points directly in Source Window
- Virtual Memory Support
  (software version 3.1 – see Appendix B)
  - Allows host software to use Expanded Memory for large programs

Chapter 2: Tutorial

Here is short tour through some of the new features in the *iceMASTER* host software. The tutorial is based primarily on the C program **TA.C** shown on the following pages.

Program TA.C

```
 1  /* ta.c */
 2
 3  /* #defines set on compilation command line:
 4  **      K_F      Keil/Franklin
 5  **      I_A      IAR/Archimedes
 6  **      ---------------------------
 7  **      LARGE    large memory model
 8  **      COMPACT  compact memory model
 9  **      SMALL    small memory model
10  **      ---------------------------
11  **      A51      8051 architecture
12  **      A11      68HC11 architecture
13  */
14
15  #ifdef  K_F          /*{*/
16  #  pragma   pl(86)       /* Listing File Page Length (lines) */
17  #  pragma   pw(132)      /* Listing File Page Width  (characters) */
18  #endif  /*K_F*/      /*}*/
19
20  #include <stdio.h>
21  /*#include <stdarg.h>*/
22  #include <string.h>
23
24  #if     defined(LARGE)       /*{{ large memory model   */
25  #  define   ARYSIZ  10
26  #elif   defined(COMPACT)     /*}{ compact memory model */
27  #  define   ARYSIZ  7
28  #elif   defined(SMALL)       /*}{ small memory model   */
29  #  define   ARYSIZ  4
30  #else
31      ??? /*(force syntax error)*/
32  #endif                       /*}} memory model         */
33
34  #   define  LIST_SIZ    3
35  #   define  TRUE        1
36
37  /************/
38  /* typedefs */
39  /************/
40  typedef unsigned char   uchar;
41  typedef unsigned int    uint;
42
43  #if     defined(K_F)                          /*{{*/
44      typedef struct List {
45          uint         list_uival;
46          struct List* list_p_nxt;
47          } LIST;
48      typedef struct List*    P_LIST;
49  #elif   defined(I_A)                          /*}{*/
50      typedef struct List*    P_LIST;
51      typedef struct List {
52          uint         list_uival;
53          P_LIST       list_p_nxt;
54          } LIST;
55  #endif  /*compiler*/                         /*}}*/
```

```
56
57                              /**********/
58                              /* main() */
59                              /**********/
60  main ()
61  {
62      float       iteration = 0.0;
63      uchar       ary[ARYSIZ];
64      uint        ai = 1, aj = 2;
65      LIST        nodes[LIST_SIZ];
66      uint        li, lj;
67      P_LIST      list_head = (P_LIST)NULL;
68      P_LIST      p_list;
69      uint        main_loop_factor = 1;
70
71      while (TRUE) {
72
73  Array_Init: /* (for demo's -- in case lines inserted/deleted) */
74          /* array demo */
75          for ( ai = 0;  ai < ARYSIZ;  ai++ ) {
76              aj = (ai * 2) + main_loop_factor;
77              ary[ai] = aj;
78              } /*for ai*/
79
80  List_Init: /* (for demo's -- in case lines inserted/deleted) */
81          /* structure/list demo */
82          /* initialize fields/links */
83          for ( li = 0;  li < LIST_SIZ;  li++ ) {
84              nodes[li].list_uival = (li * 4) + main_loop_factor;
85              nodes[li].list_p_nxt = (li == 0) ? NULL : &(nodes[li-1]) ;
86              } /*for li*/
87          list_head = &(nodes[LIST_SIZ-1]);
88
89  List_Scan: /* (for demo's -- in case lines inserted/deleted) */
90          /* scan through nodes in list */
91          p_list = list_head;
92          while (p_list != NULL) {
93              lj     = p_list->list_uival;
94              p_list = p_list->list_p_nxt;
95              } /*while p_list*/
96
97          iteration += 1.1;
98          main_loop_factor++;
99
100 End_Loop: ; /* (for demo's -- in case lines inserted/deleted) */
101
102         } /*while TRUE*/
103
104 } /*main*/
```

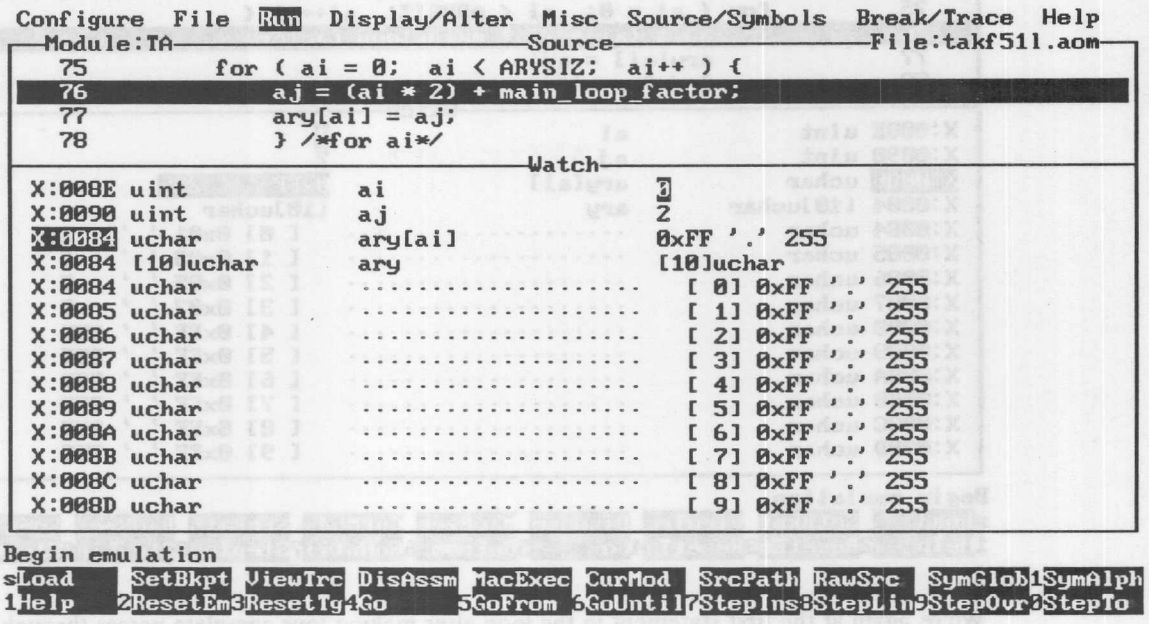
The program has two parts. The first part is a loop to initialize the elements in an array of **unsigned chars**. The second part uses an array of **structures** to implement a singly-linked list. The first sub-loop in this second part establishes the linkage through the nodes in the list. The second sub-loop traverses the linked list using a pointer variable.

We have already loaded the program TAKF51L.AOM into the *iceMASTER* emulator (TAKF51L.AOM: Test/program A, Keil/Franklin compiler, MCS-51 chip architecture, Large memory model, Absolute Object Module format). We have also opened the Watch Window and set several Watch Expressions:

```
Configure  File  Run  Display/Alter  Misc  Source/Symbols  Break/Trace  Help
Module:TA                          Source                    File:takf51l.aom
  75          for ( ai = 0;  ai < ARYSIZ;  ai++ ) {
  76              aj = (ai * 2) + main_loop_factor;
  77              ary[ai] = aj;
  78              } /*for ai*/
                                    Watch
X:008E uint              ai                    1
X:0090 uint              aj                    2
X:0085 uchar             ary[ai]               0xFF '.' 255
X:0084 [10]uchar         ary                   [10]uchar
X:0084 uchar             ..................      [ 0] 0xFF '.' 255
X:0085 uchar             ..................      [ 1] 0xFF '.' 255
X:0086 uchar             ..................      [ 2] 0xFF '.' 255
X:0087 uchar             ..................      [ 3] 0xFF '.' 255
X:0088 uchar             ..................      [ 4] 0xFF '.' 255
X:0089 uchar             ..................      [ 5] 0xFF '.' 255
X:008A uchar             ..................      [ 6] 0xFF '.' 255
X:008B uchar             ..................      [ 7] 0xFF '.' 255
X:008C uchar             ..................      [ 8] 0xFF '.' 255
X:008D uchar             ..................      [ 9] 0xFF '.' 255
Begin emulation
sLoad    SetBkpt  ViewTrc  DisAssm  MacExec  CurMod   SrcPath  RawSrc   SymGlob1SymAlph
1Help    2ResetEm3ResetTg4Go      5GoFrom 6GoUntil7StepIns8StepLin9StepOvr0StepTo
```

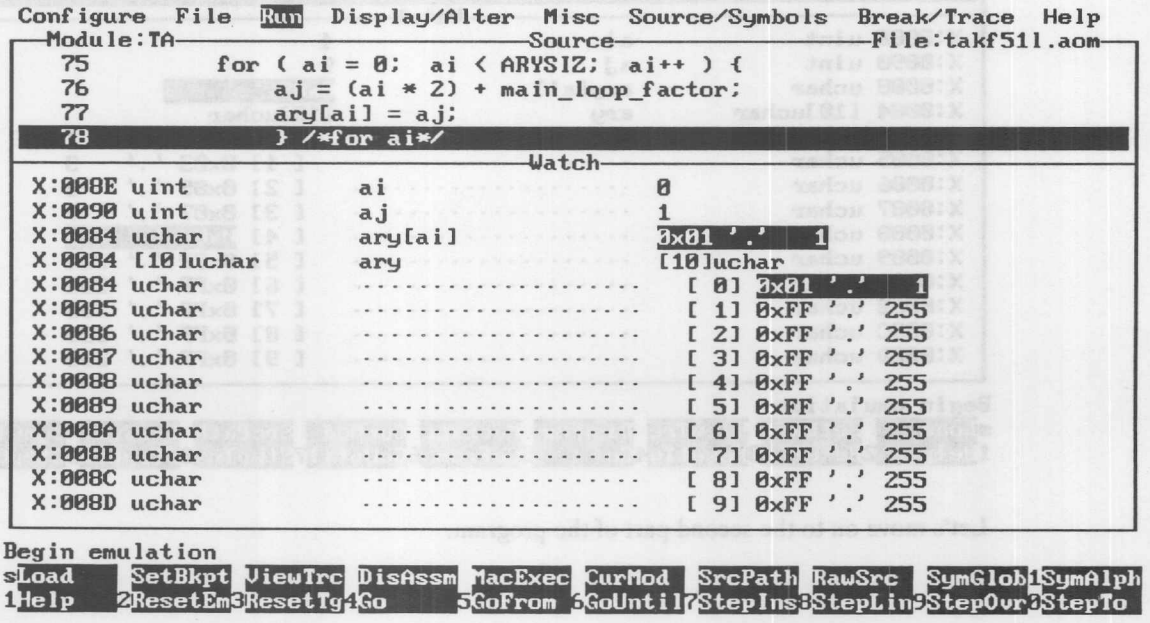
The program is about to step through the array *ary* to initialize each element. There are four Watch Expressions in the Watch Window. *ai* and *aj* are simple **unsigned integer** variables at locations 0x008E and 0x0090, respectively, in External Data memory. Their current values are 1 and 2 (from source line #64). The third Watch Expression, *ary[ai]* monitors a single element in the array *ary*, but the element being monitored changes as the value of the subscript *ai* changes. The fourth Watch Expression, *ary*, watches the entire array. The "value" shown for *ary* is its data type: "[10]uchar", meaning that it is an array of 10 elements, each of which is an **unsigned char**. Following this header line for the array, each element in the array appears. The current value in each element is displayed in hexadecimal, character and unsigned decimal formats, preceded by the element's actual subscript.

Let's single-step the program by one HLL (Higher-Level Language) source line:



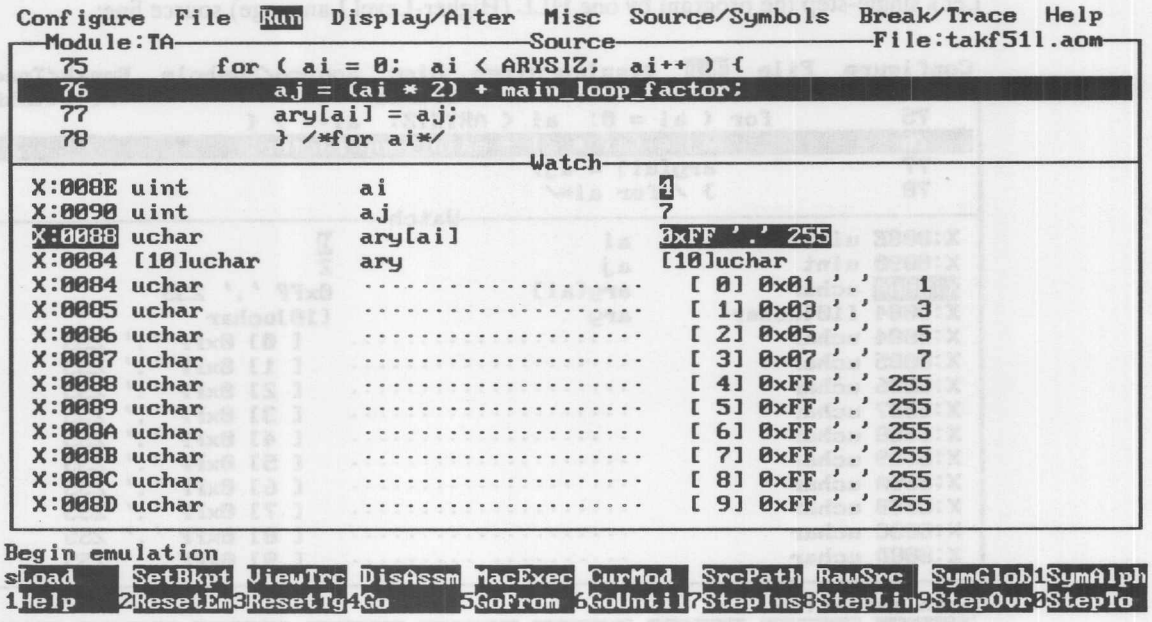
The program just executed the **for** loop initialization statement (*ai = 0;*) and is about to execute the first line in the **for** loop. Notice that the value of *ai* has changed from 1 to 0; it is highlighted to notify you that the value changed. Also note that the address for the Watch Expression *ary[ai]* has changed (because the subscript value *ai* changed); it, too, is highlighted to flag the change.

Now, lets step twice to get past the first array element assignment at line #77 at the end of the loop:

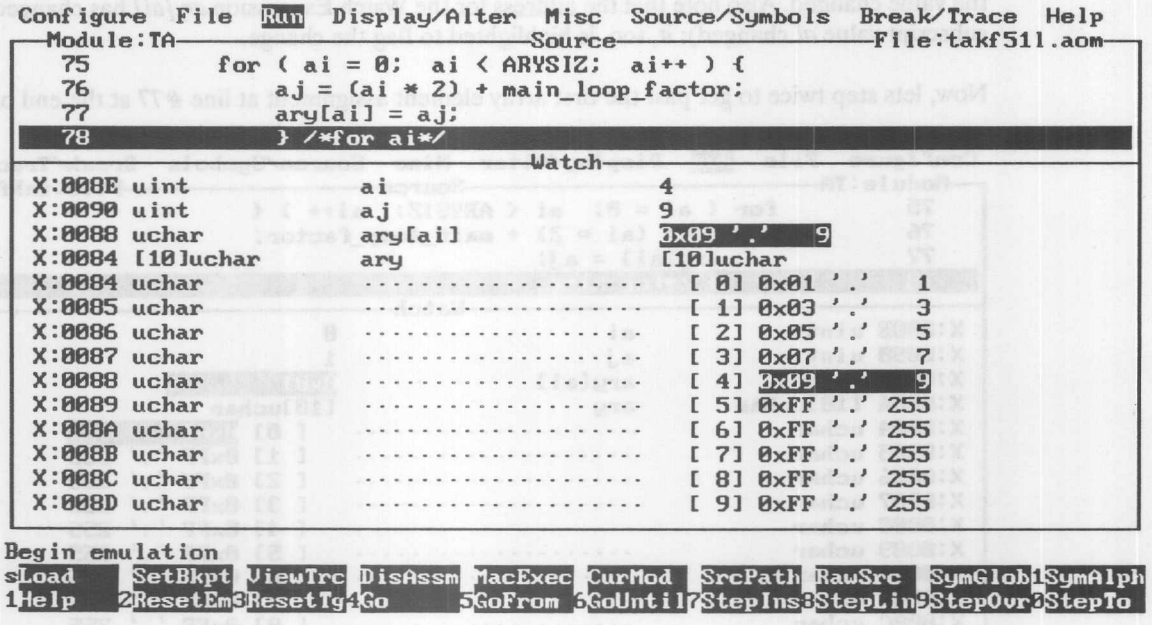


After executing the assignment *ary[ai] = aj;*, we see that the value stored in the first element (subscript value zero) has changed, and the corresponding change is reflected in the copy being monitored by Watch Expression *ary[ai]*.

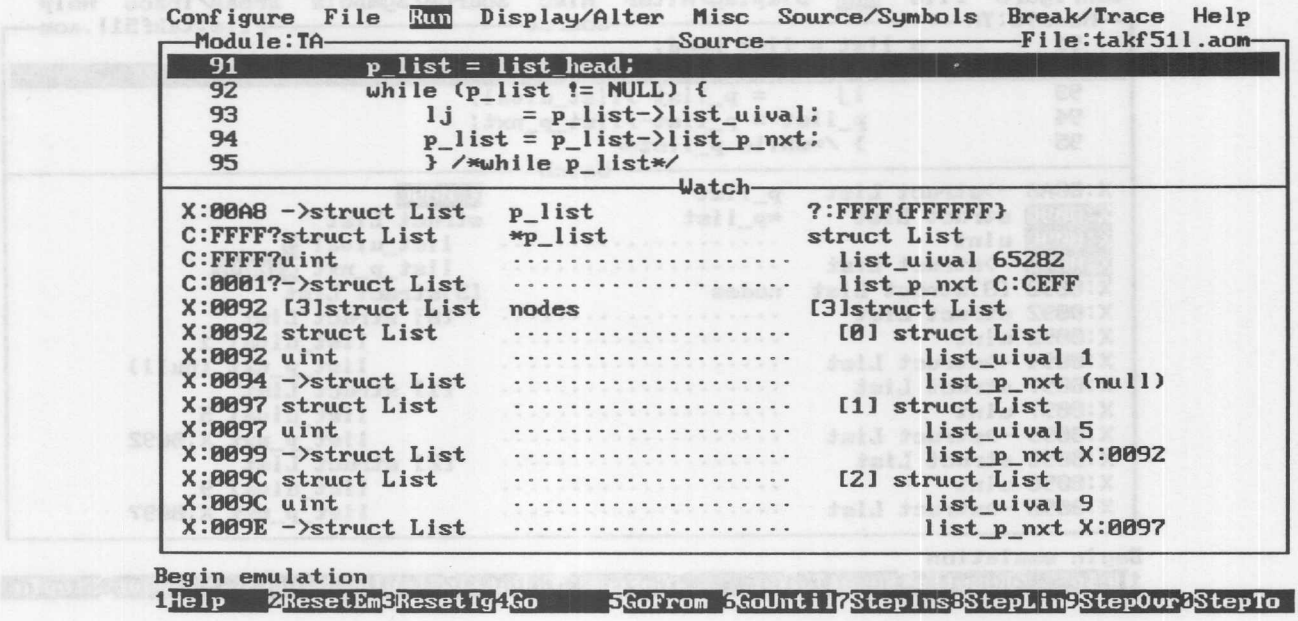
Let's take several more trips through the loop:



We're again at the first statement in the loop after making four complete passes through the loop. The first four elements in the array have been initialized and the subscript *ai* is set for the fifth element in the array. Now let's step to the end of the loop one more time:



Let's move on to the second part of the program.



At this point, the program has executed the first sub-loop that establishes the list linkage through the **structure** nodes in the array *nodes* (an array of **structures**). The list head pointer, *list_head*, has been set to point at the first node in the list. That node is the last element in the array (subscript value [2], at address 0x009C in External Data memory), as the list was linked "backwards" through the array elements. The list is linked using the *list_p_nxt* field in each node.

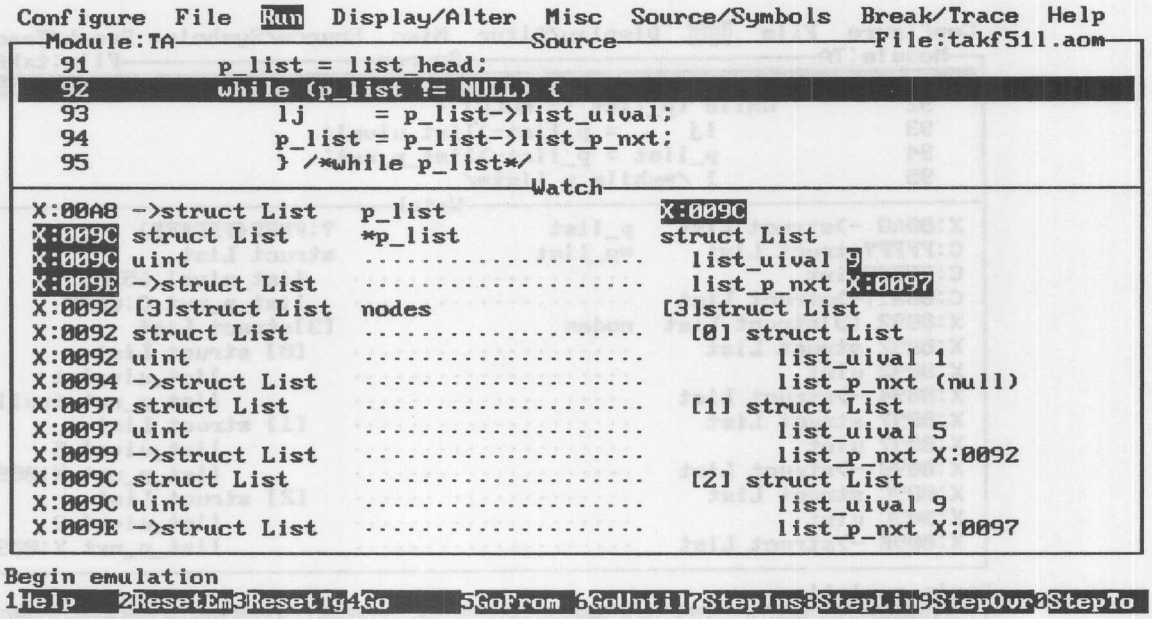
The Watch Window contains three Watch Expressions.

The variable *p_list* is a pointer to a LIST node and it currently contains an illegal pointer value. That's fine because the program has not yet assigned a value to *p_list*. A pointer such as *p_list* occupies 3 bytes in memory. The first byte contains a selector value that indicates which memory space (e.g, Code memory, External Data memory, etc.) is being referenced by the rest of the pointer. The other two bytes in the pointer contain the offset into that memory space. In this case, the memory space selector value is invalid (0xFF), so the software displays it's code as a question mark (**?**) followed by a colon (**:**) and the offset bytes in hexadecimal. In addition, because the pointer is illegal, all three "raw" bytes in the pointer's value are displayed in hexadecimal within braces.

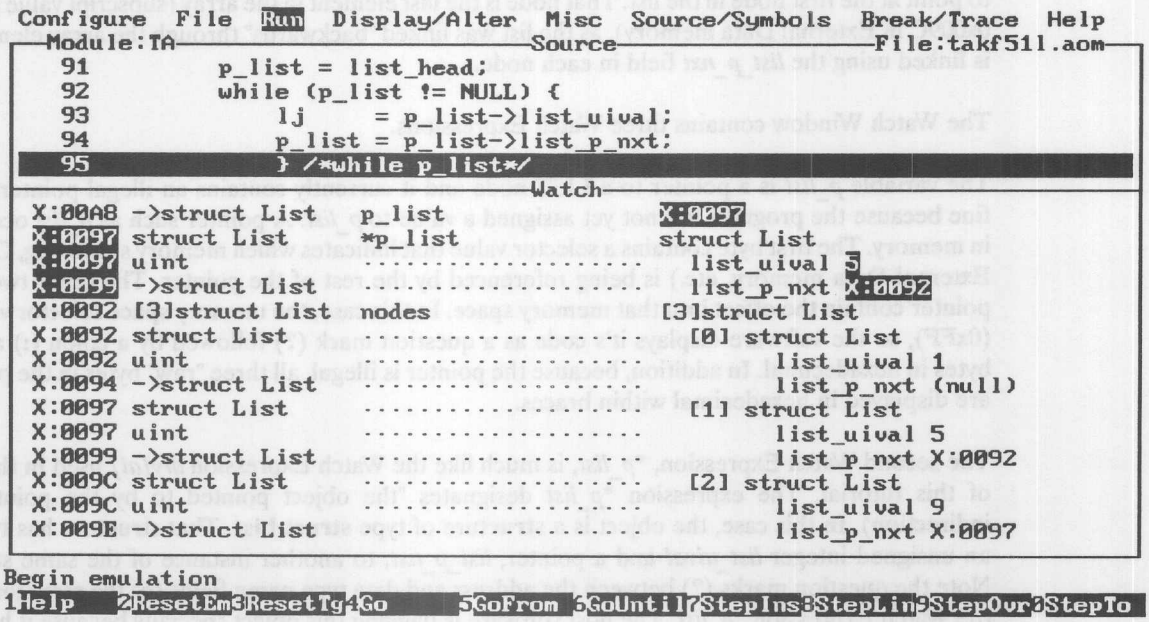
The second Watch Expression, **p_list*, is much like the Watch Expression *ary[ai]* used in the earlier part of this tutorial. The expression **p_list* designates "the object pointed to by the pointer *p_list*" (C indirection). In this case, the object is a **structure** of type **struct List**. That structure has two members: an unsigned integer *list_uival* and a pointer, *list_p_nxt*, to another instance of the same structure type. Note the question marks (?) between the address and data type name fields for the **structure** denoted by the Watch Expression **p_list*. The host software is flagging this object specially because it had to "follow" an invalid or NULL pointer to get to the object.

The third Watch Expression is *nodes*, which is the array of **structures** containing the nodes in the linked list.

Let's execute the highlighted line to initialize *p_list* so that it points at the first node in the chain:

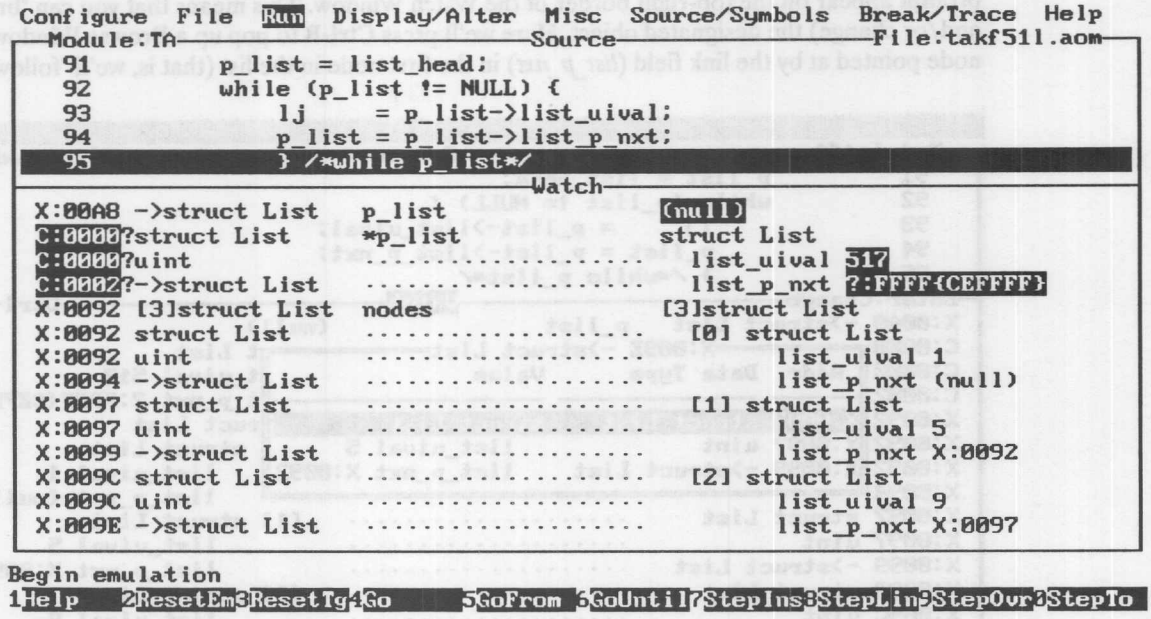


Notice that *p_list* now contains a valid pointer value and that the node denoted by the expression **p_list* is the first node in the linked list. Let's step through to the end of the first pass in this loop:



p_list now points at the second node in the chain and the updated Watch Expression **p_list* shows a highlighted copy of that node.

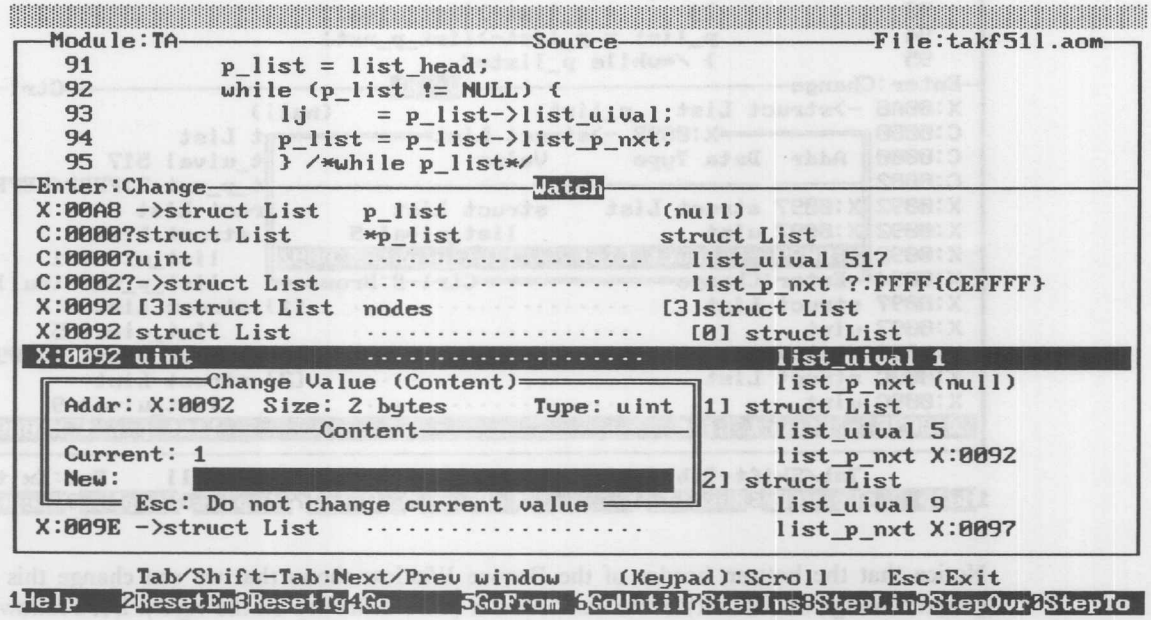
Finally, let's step until very end of the loop:



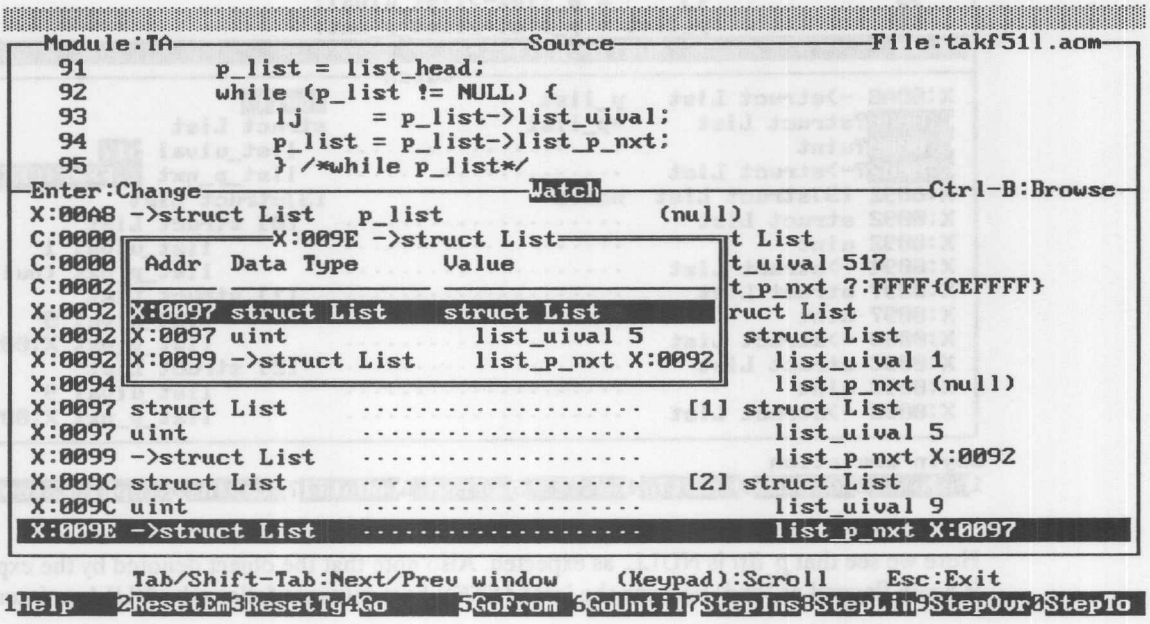
Here we see that *p_list* is NULL, as expected. Also note that the object denoted by the expression **p_list* is again flagged as "bad" because the *iceMASTER* host software followed a NULL pointer to get to it.

Browsing

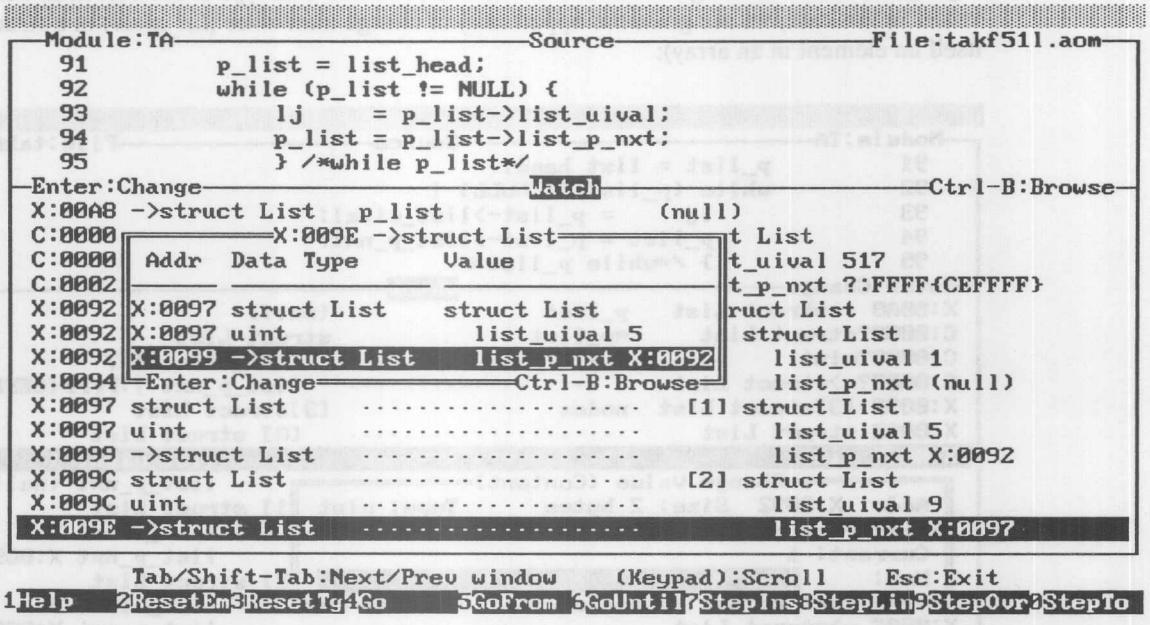
You can "jump into" the Watch Window to browse/inspect/change the values in objects designated by the Watch Expressions. To do this, select the *Configure|Windows|Goto* command (shortcut: press the **Tab** key). As you scroll up or down, if the currently highlighted item is a scalar (integral or floating-point) or a pointer, you will see the **Enter:Change** prompt appear on the top-left border of the Watch Window. When you press the **Enter** key, a Change Box pops up to allow you to change the value stored in the highlighted item. A Change Box shows the address, size and data type of the object, as well as its current value. In this case, we are given the opportunity to change a member (*list_uival*) in a structure (which is itself an element in an array):



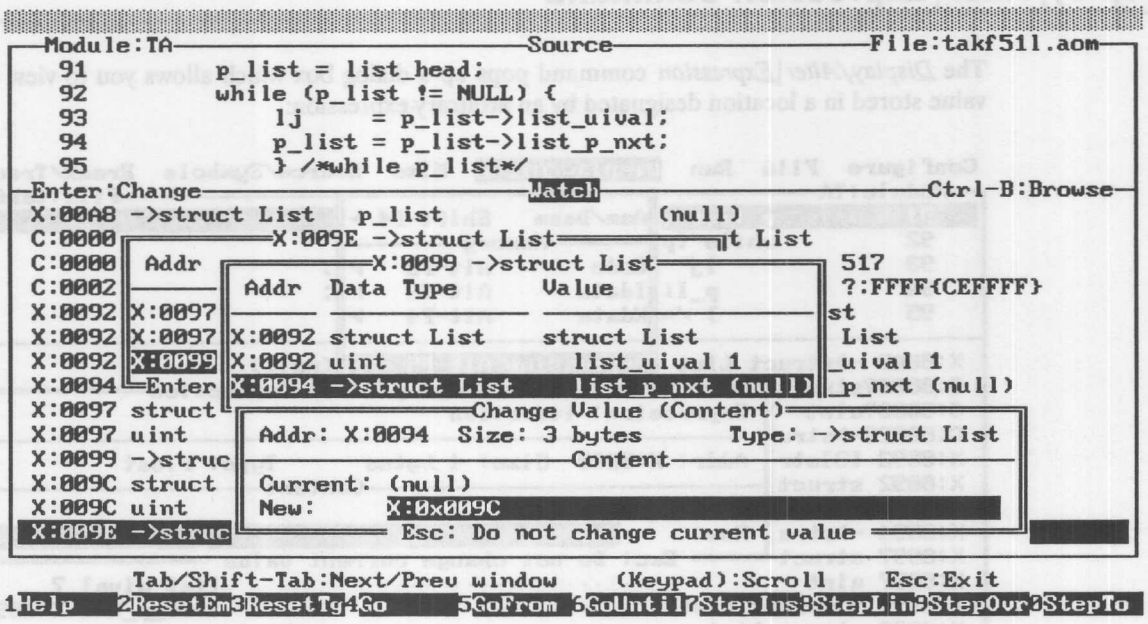
If the currently highlighted item is a pointer, structure, union or array, you will see the **Ctrl-B:Browse** prompt appear on the top-right border of the Watch Window. This means that you can "browse" (inspect and/or change) the designated object. Here we'll press **Ctrl-B** to pop up a Browse Window to look at the node pointed at by the link field (*list_p_nxt*) in the first node in the list (that is, we'll "follow" the pointer):



A Browse Window shows the entire object denoted by the pointer. A Browse Window is scrollable, moveable and resizeable. Thus, you can inspect large objects with no problem. You can even write the entire contents of a Browse Window to a file of your choice. Let's move the highlight bar to the link field (*list_p_nxt*) in the **struct** we are inspecting:

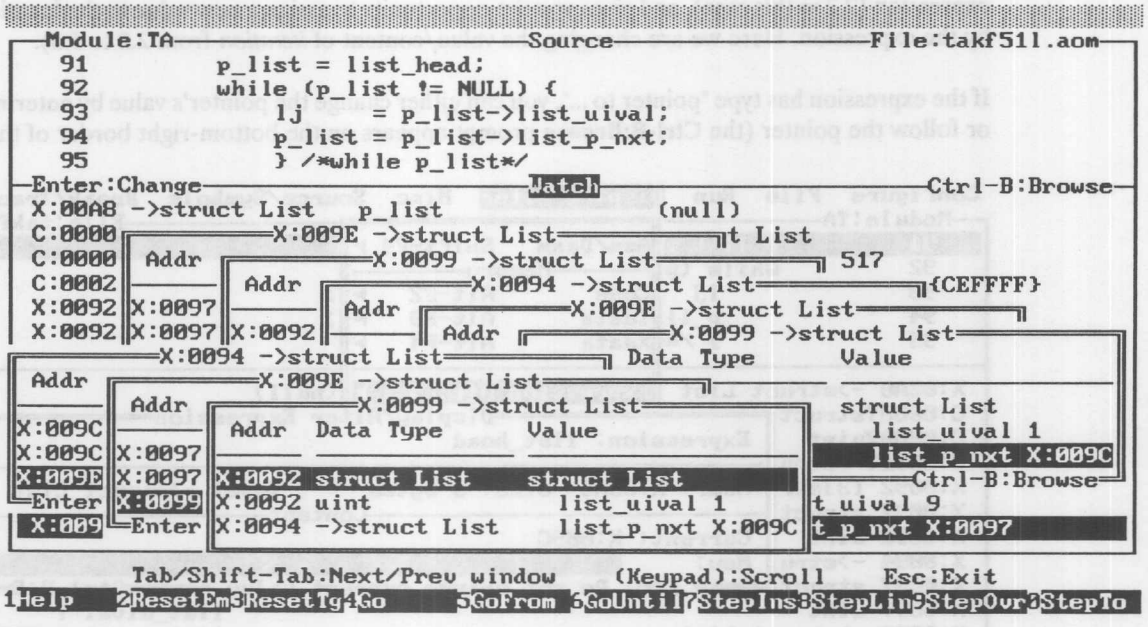


Notice that the bottom border of the Browse Window shows that we can change this pointer value (**Enter:Change** on the left) or we can follow it (**Ctrl-B:Browse** on the right). Let's follow it to the next node in the list.



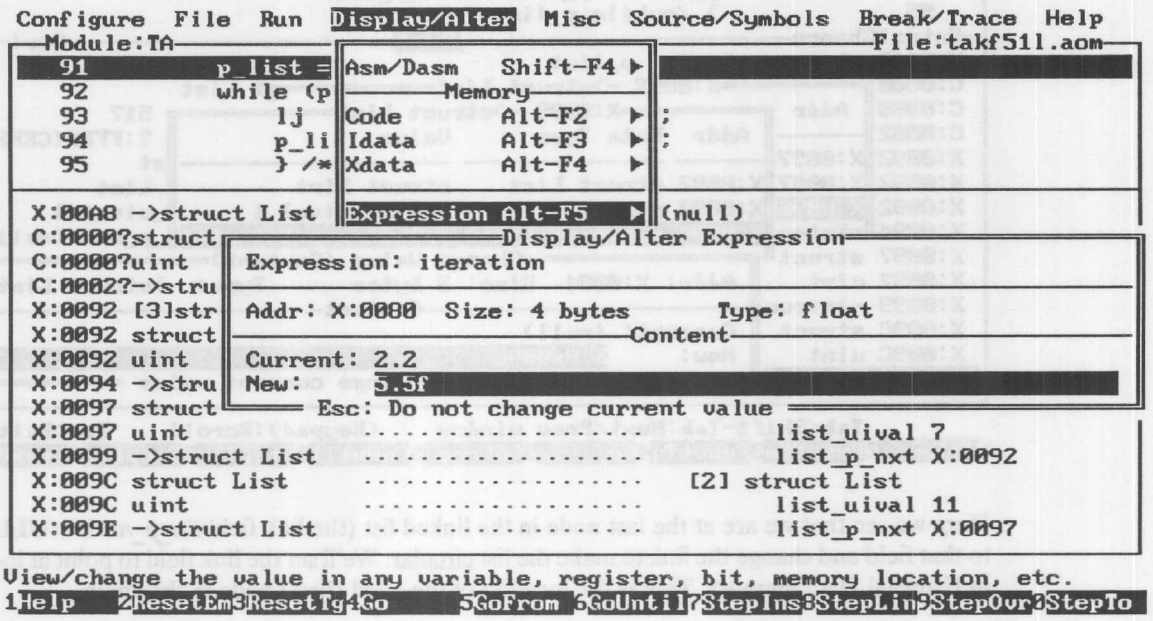
Here we see that we are at the last node in the linked list (the link field *list_p_nxt* is NULL). Let's move to that field and change the link to make the list circular. We'll set the link field to point at location 0x009C in External Data memory. That is the location of the head (first) node in the chain.

Now if we continue to follow the link field (repetitively pressing **Ctrl-B** while the highlight bar is on the member *list_p_nxt*), we'll go on, and on, and on ...



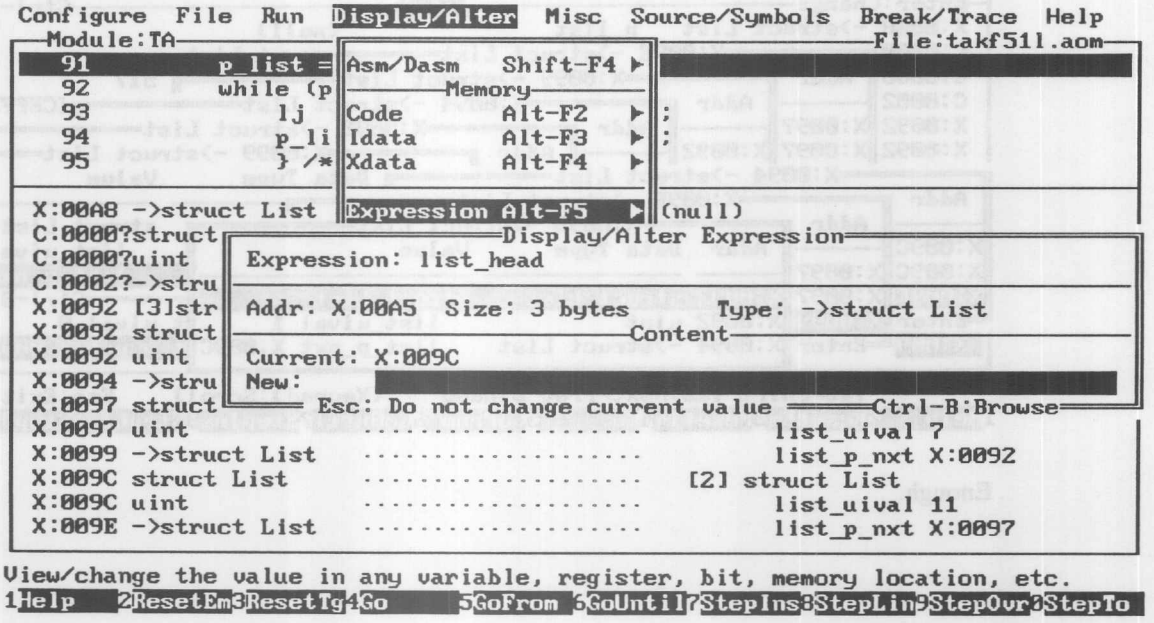
Enough.

The *Display/Alter|Expression* command pops up a dialog box which allows you to view or change the value stored in a location designated by an arbitrary expression:

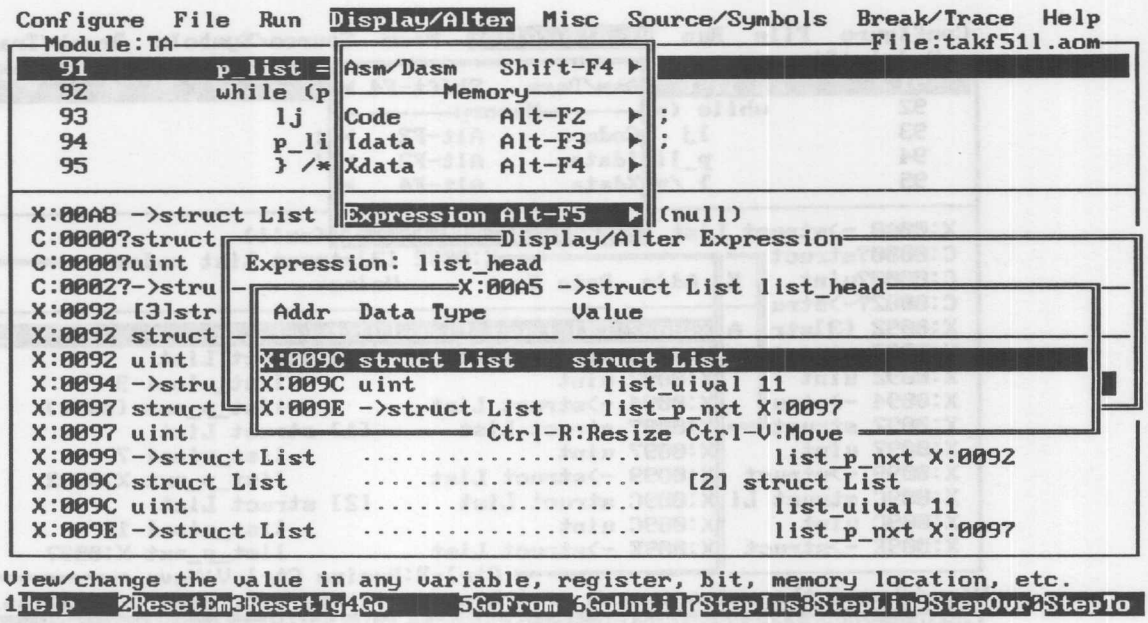


The dialog box shows the address, size (in bytes or bits) and data type of the object denoted by the expression you type (in this case, the variable *iteration*). Additionally, it shows the current value of the expression (2.2 in this case), and gives you the opportunity to assign a new value to the location designated by the expression. Here we are changing the value/content of *iteration* from 2.2 to 5.5).

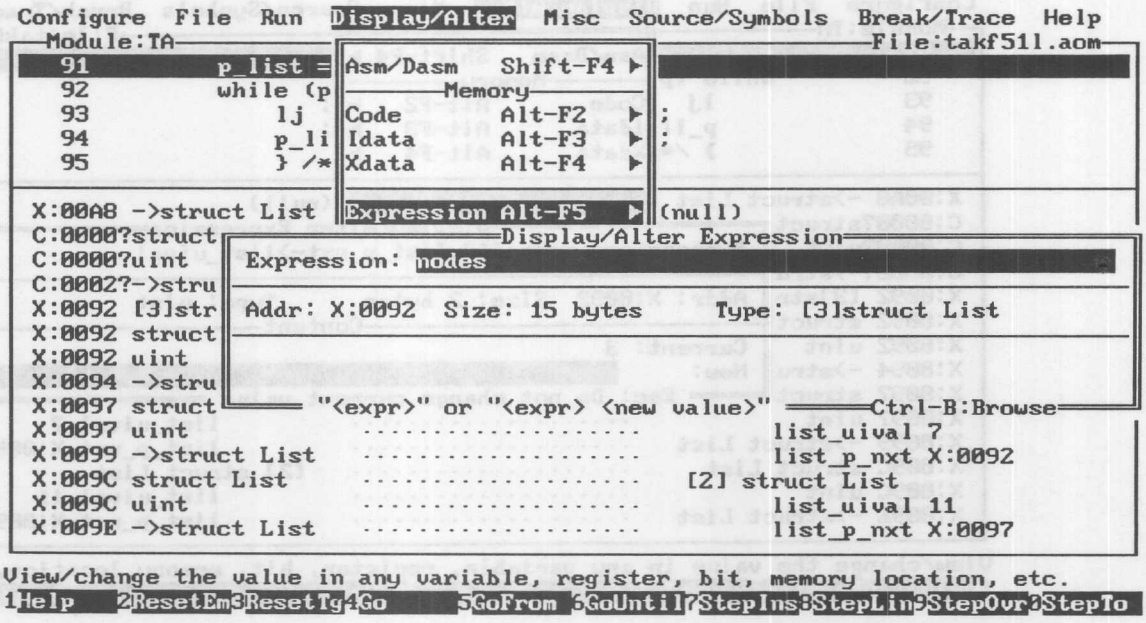
If the expression has type "pointer to ...", you can either change the pointer's value by entering a new value or follow the pointer (the **Ctrl-B:Browse** prompt appears on the bottom-right border of the box):



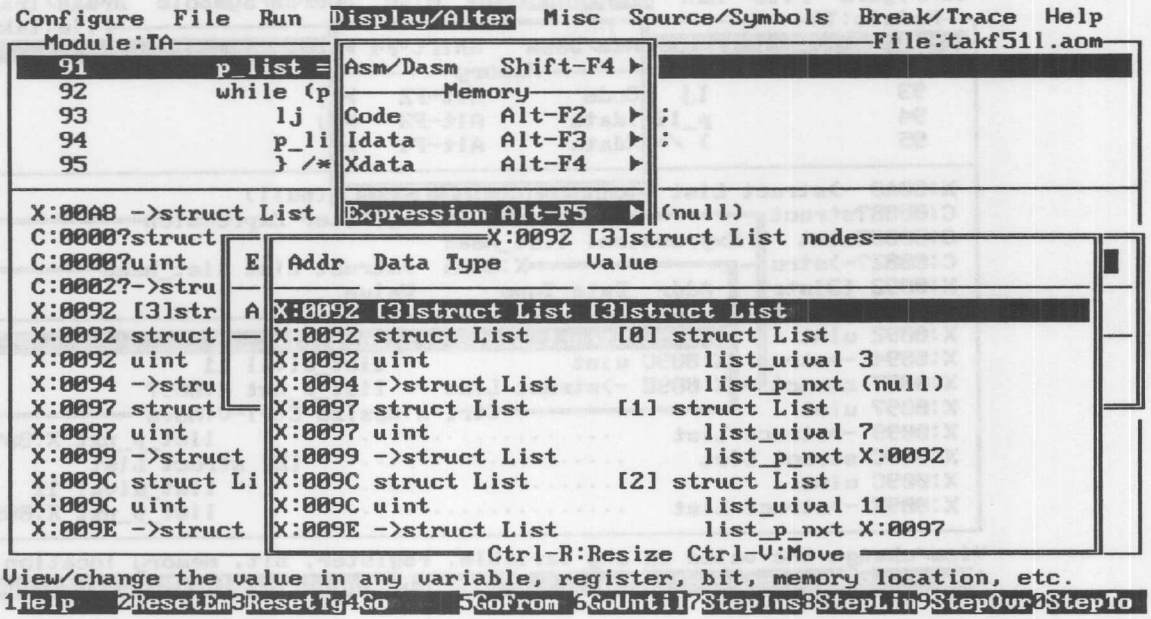
If we choose to follow it by pressing **Ctrl-B**, again a Browse Window pops up to let us do just that:



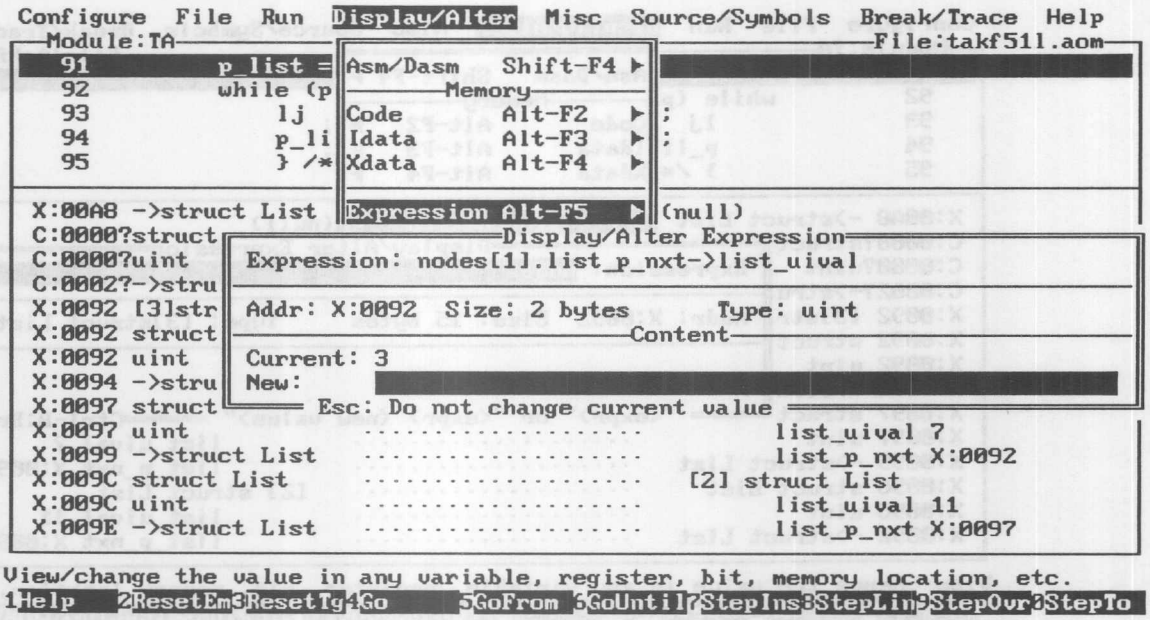
If the expression has type "array of ...", the *Display/Alter|Expression* command shows us the size of the entire array *nodes* (15 bytes in this) case and prompts us that we can browse (inspect) the array (**Ctrl-B:Browse** prompt on the bottom-right border of the box):



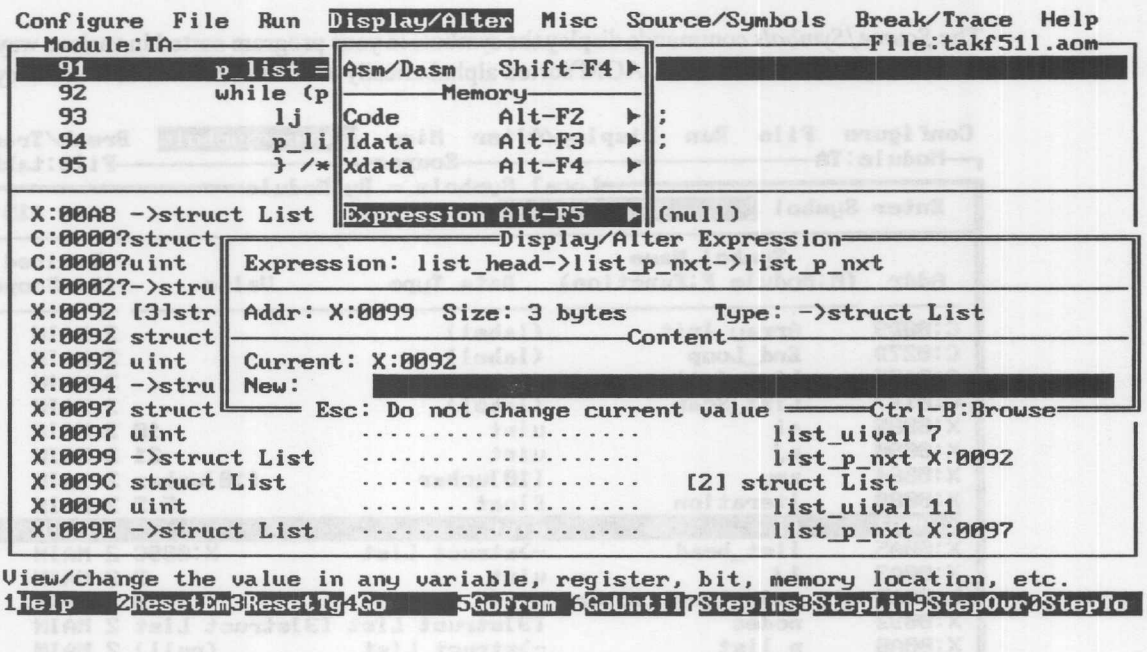
Pressing **Ctrl-B** pops-up a Browse Window, letting us inspect and/or change the individual elements in the array:



You can enter almost any expression in the *Display/Alter|Expression* dialog box, designating array elements and **struct/union** members just as you would in your C program:

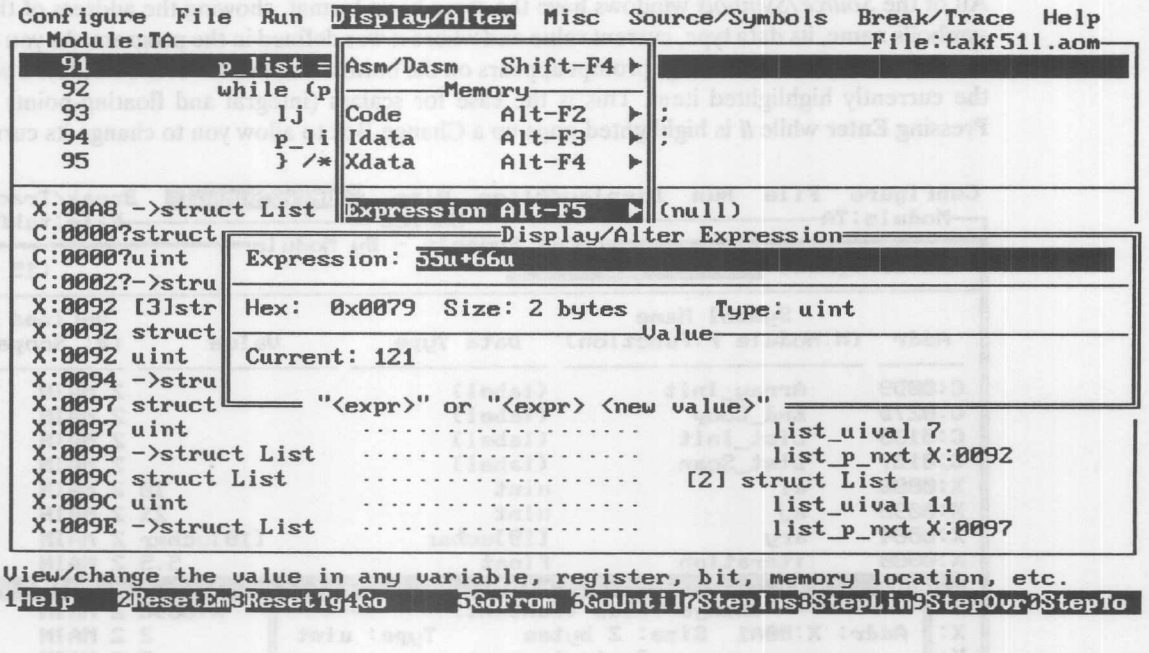


Or you can follow a pointer through several levels of indirection in one step:

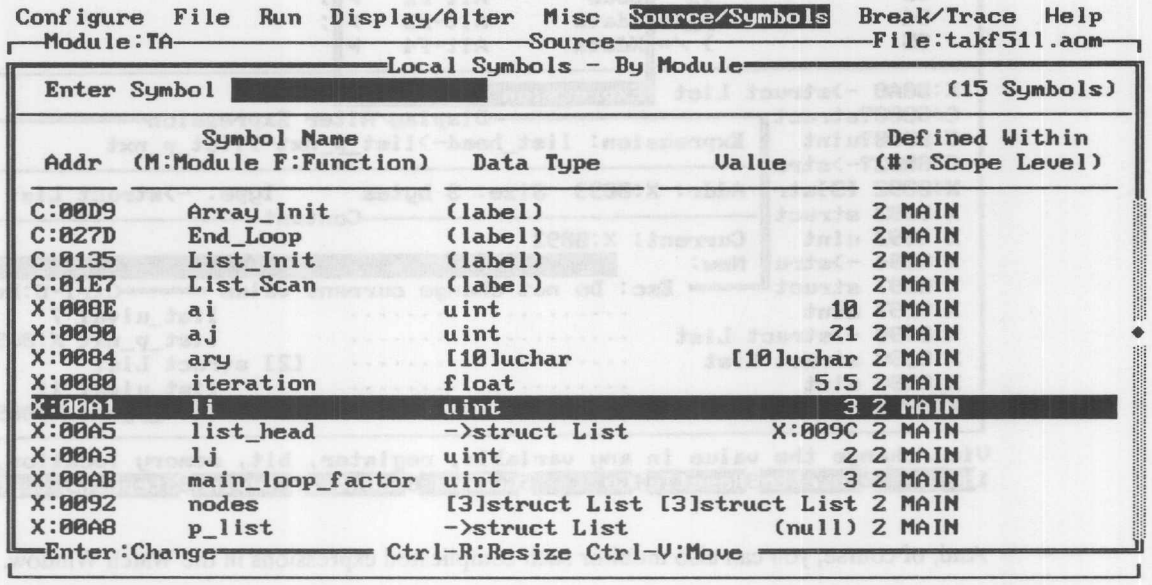


And, of course, you can also monitor such complicated expressions in the Watch Window.

Finally, in the simplest case, the *Display/Alter|Expression* command even operates as a calculator:

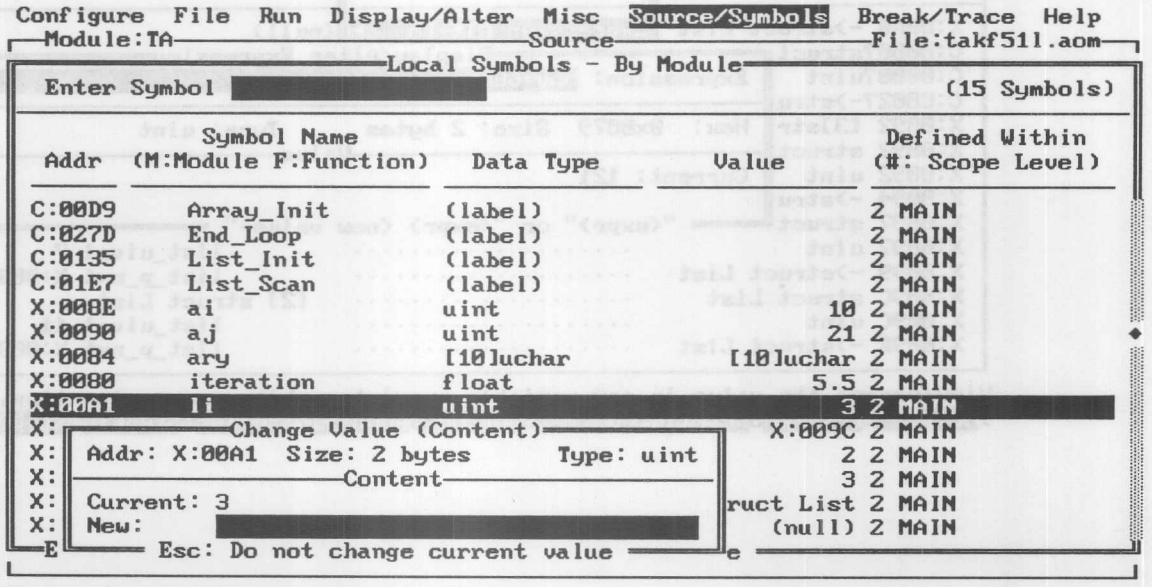


The *Source/Symbols* commands display the symbols in your program sorted in various ways. Here we see the local symbols in TAKF51L.AOM sorted alphabetically within the module in which they were defined:



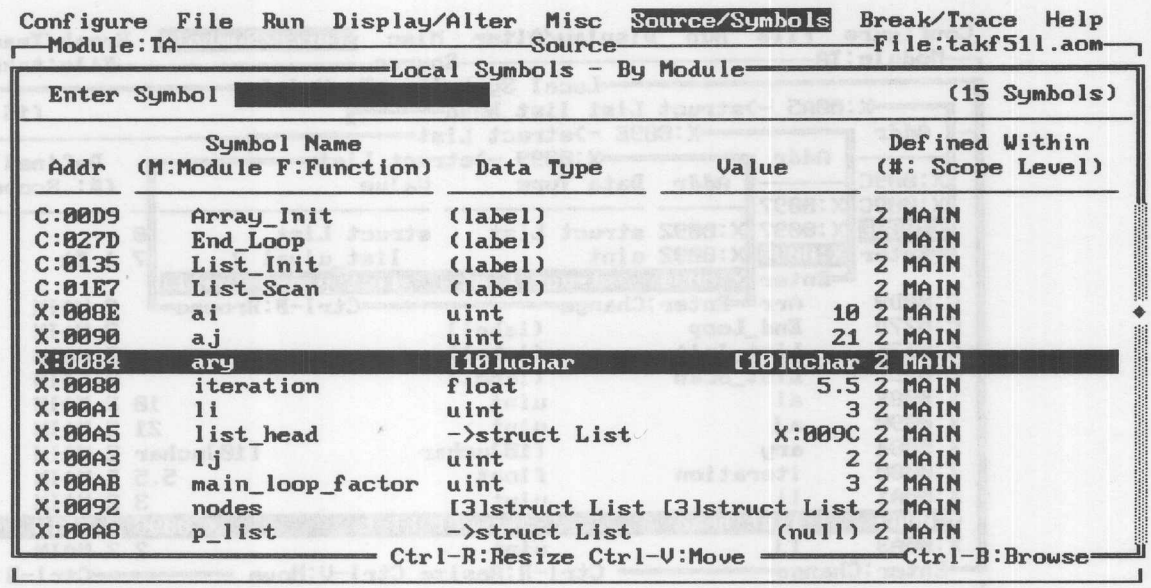
Source level debug and symbol displays

All of the *Source/Symbols* windows have the same basic format, showing the address of the symbol, the symbol's name, its data type, current value and where it was defined in the program. As you scroll through the window, the **Enter:Change** prompt appears on the bottom-left border of the window if you can change the currently highlighted item. This is the case for scalars (integral and floating-point) and pointers. Pressing **Enter** while *li* is highlighted pops up a Change Box to allow you to change its current value:



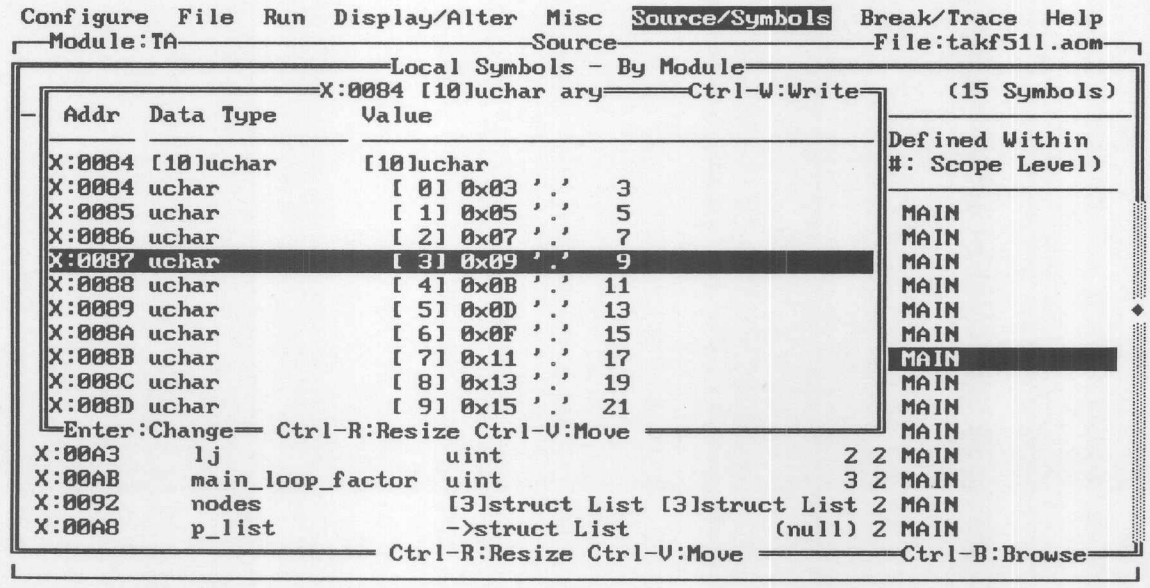
Source level debug and symbol displays

If a symbol is an array, structure or union, the "value" shown in the *Source/Symbols* windows is the symbol's data type:



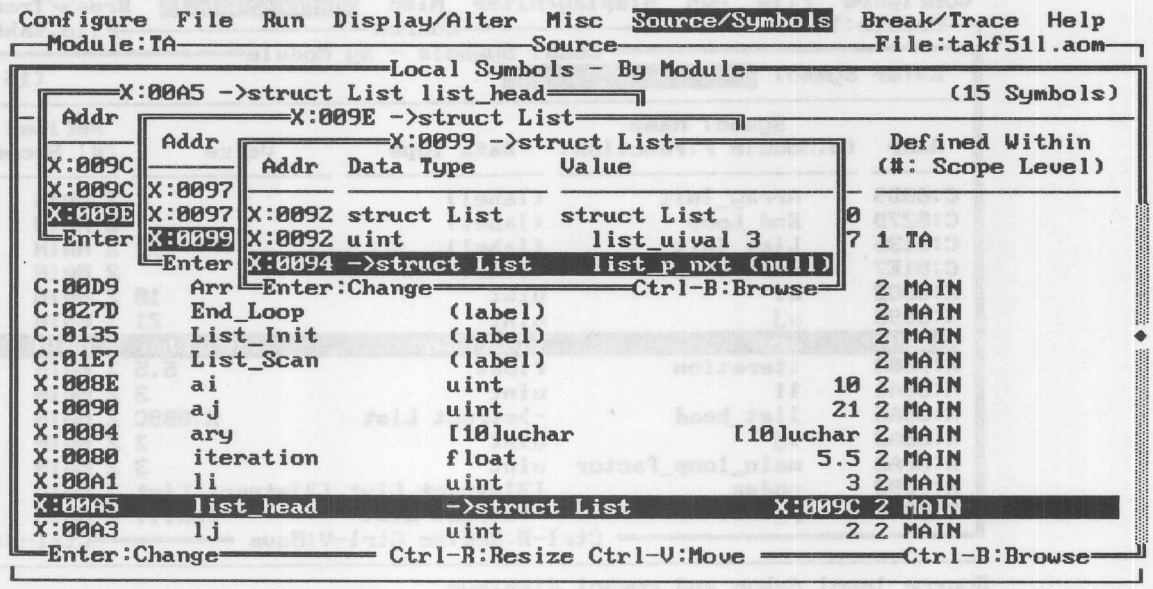
Source level debug and symbol displays

Notice that the **Ctrl-B:Browse** prompt appears on the bottom-right border of the window. You can browse (inspect) structures, unions, arrays and pointers in all the *Source/Symbols* windows. Pressing **Ctrl-B** here pops up a Browse Window, allowing us to inspect the individual elements in the array *ary*:



Source level debug and symbol displays

Similarly, if the currently-highlighted symbol is a pointer to something, we can press the **Ctrl-B** key to browse that "something". And, of course, if that "something" contains elements that are browseable, we can inspect or change them, too:

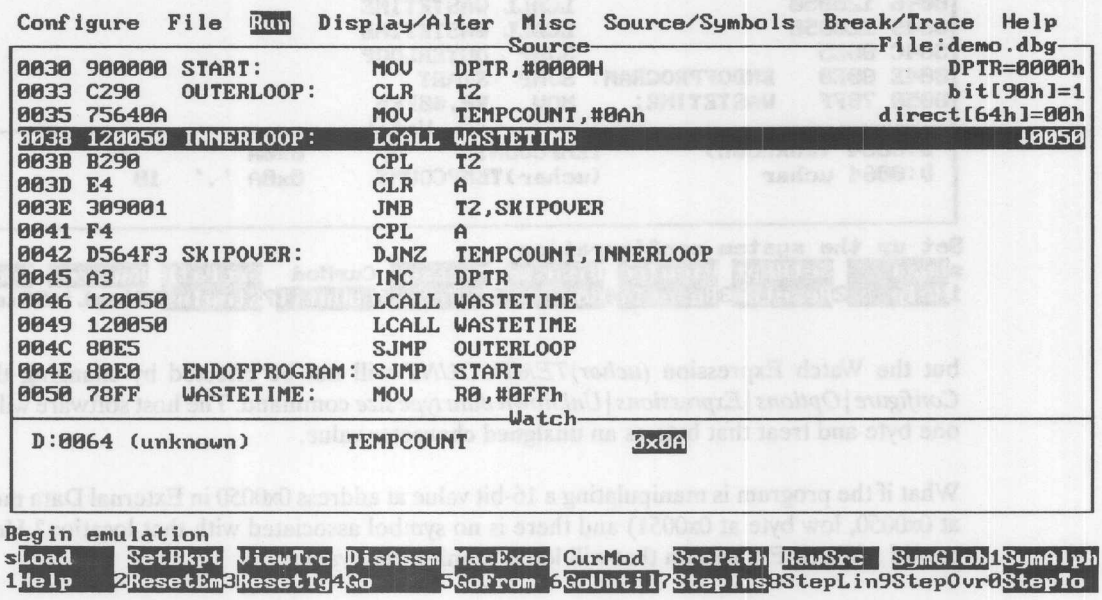


Source level debug and symbol displays

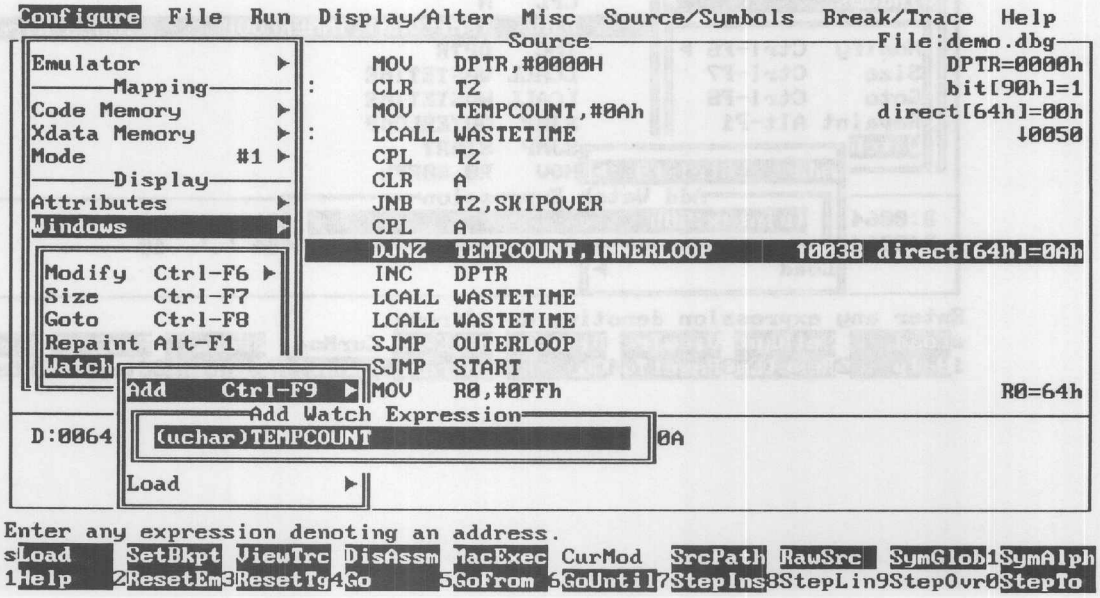
Improved Support for "Unsupported" Programs

Even if your program contains limited data type information, or no data type information at all, you can still benefit from some of the new features in the *iceMASTER* software.

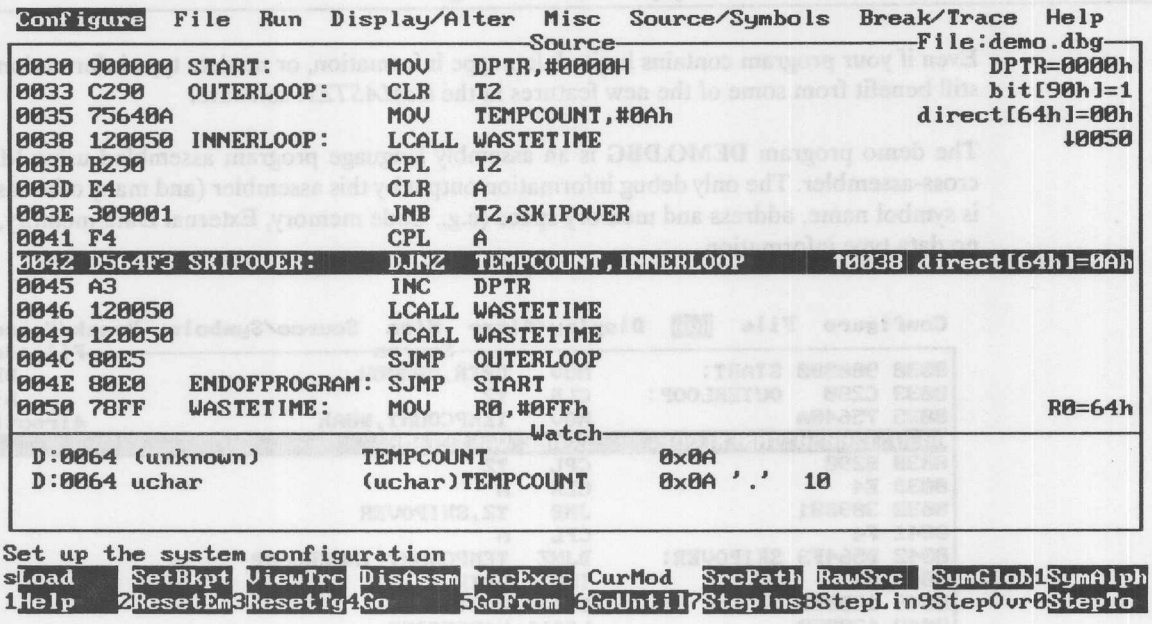
The demo program **DEMO.DBG** is an assembly language program assembled using MetaLink's 8051 cross-assembler. The only debug information output by this assembler (and many other assemblers, too) is symbol name, address and memory space (e.g., Code memory, External Data memory, etc.). There is no data type information.



In the picture above, we have loaded **DEMO.DBG** into the emulator and added the Watch Expression *TEMPCOUNT* to the Watch Window. Notice that the data type is "(unknown)". In such cases, the host software defaults to displaying the value in hexadecimal only, and the software must be "told" how many bytes to read (via the *Configure|Options|Expressions|Unknown data type size* command). Below, we are about to add a Watch Expression that will cause the host software to treat *TEMPCOUNT* as "true" one-byte **unsigned char**acter value.

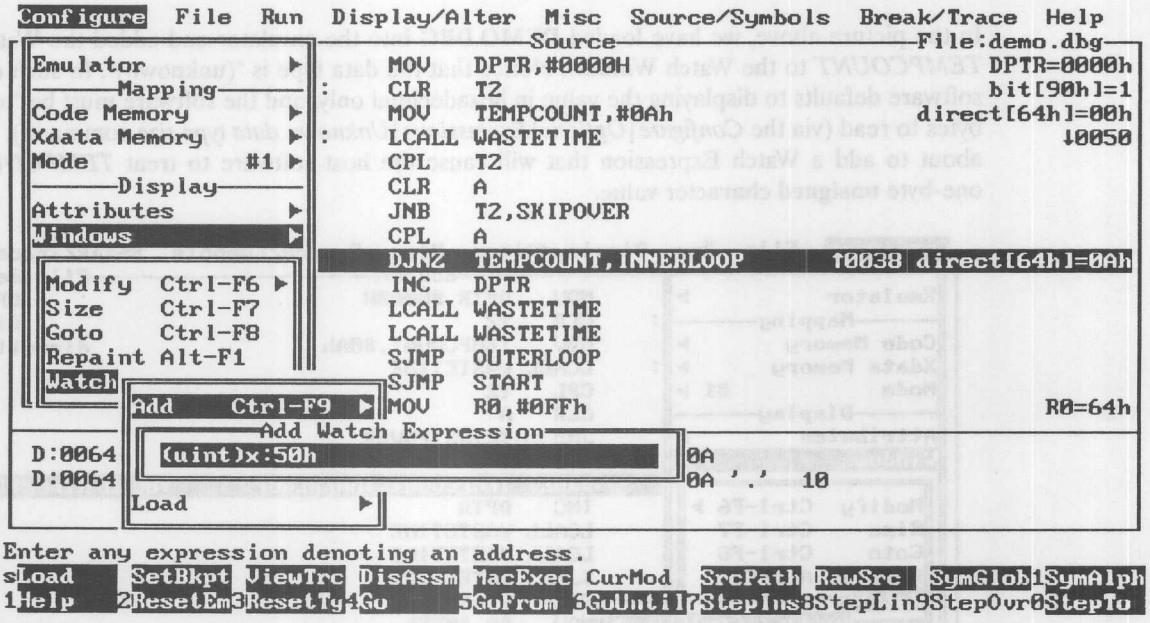


As you can see below, there isn't much difference in the display in this case:

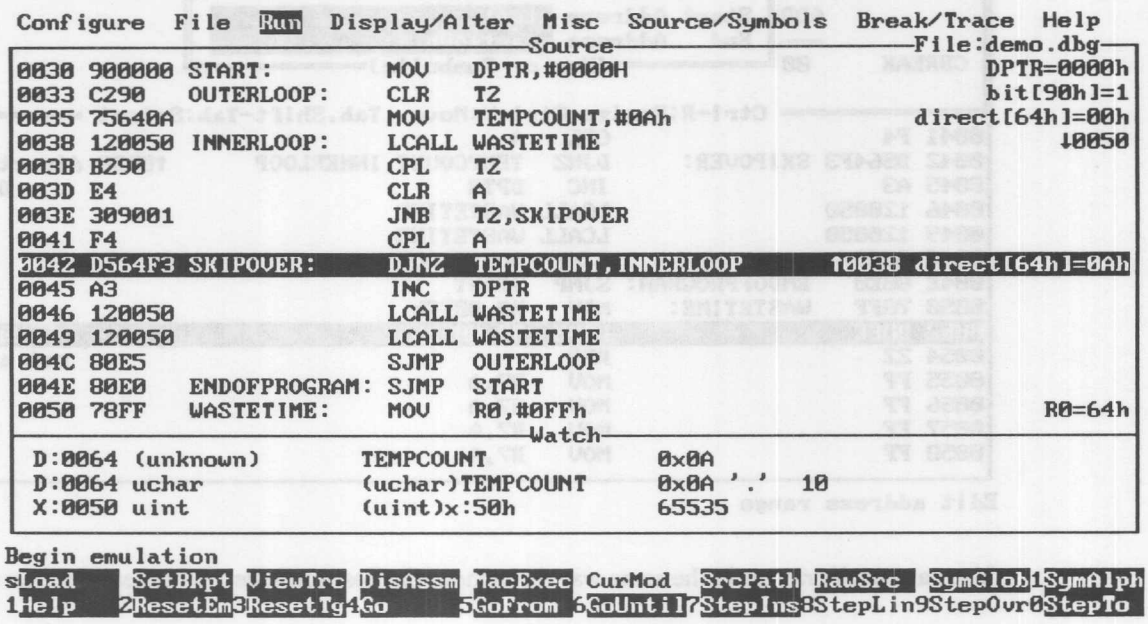


but the Watch Expression *(uchar)TEMPCOUNT* will not be effected by changing the value for the *Configure | Options | Expressions | Unknown data type size* command. The host software will always read just one byte and treat that byte as an **unsigned char**acter value.

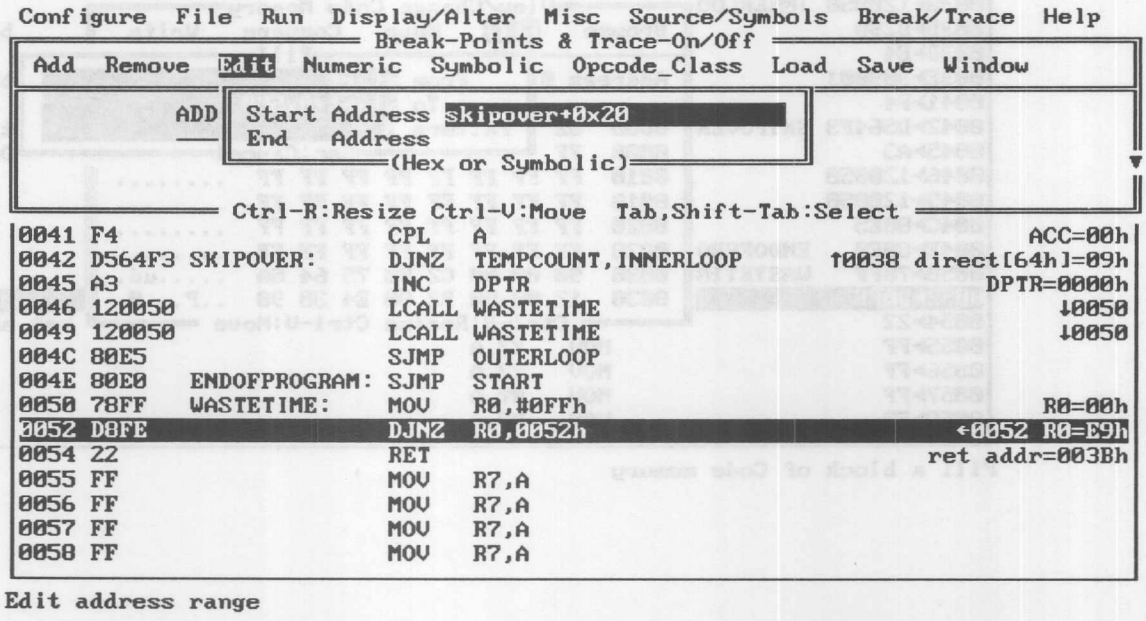
What if the program is manipulating a 16-bit value at address 0x0050 in External Data memory (high byte at 0x0050, low byte at 0x0051) and there is no symbol associated with that location? Here we are about to add a Watch Expression that will let us monitor that value:



After the Watch Expression *(uint)x:50h* is added, the host software will monitor the two bytes starting at location 0x0050 in External Data memory and treat those two bytes combined as a 16-bit **unsigned int**eger value. The default display format for such values is (unsigned) decimal notation, although you can change that to your choice of character, decimal or hexadecimal notations using the *Configure|Options|Expressions|int/uint* command:

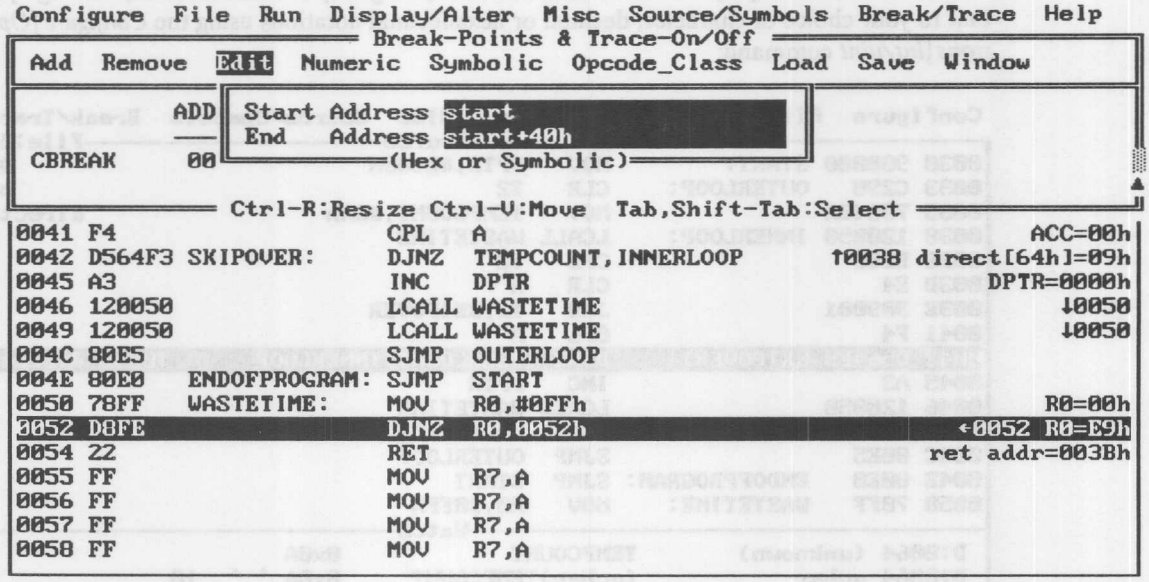


With the ability throughout the host software to specify addresses using expressions rather just simple labels or absolute addresses, you can, for example, set break-points relative to a label:



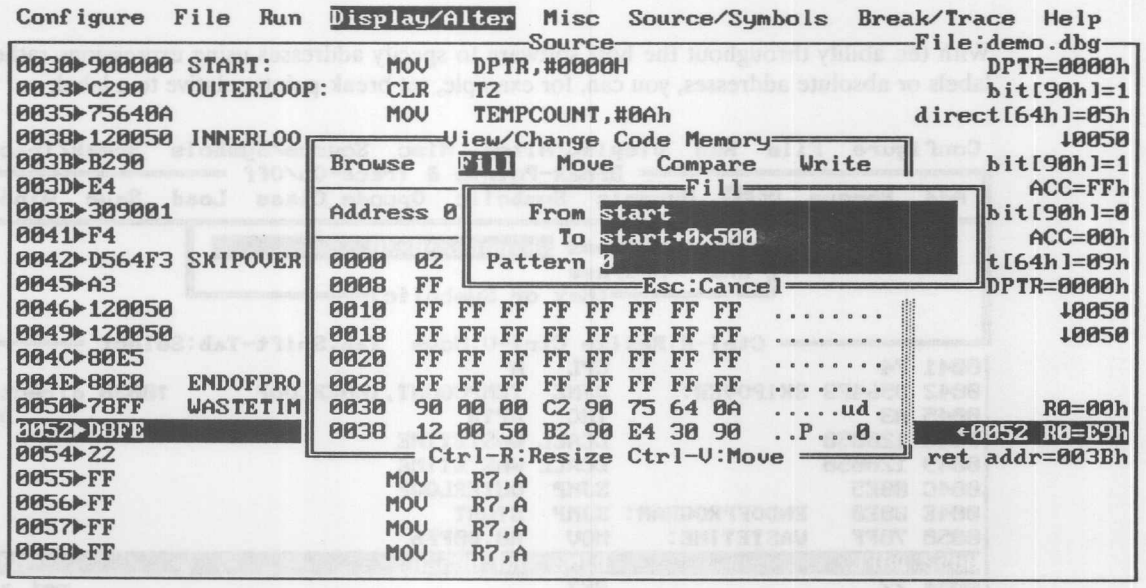
This is handy if you have a multi-module assembly language program and are using the assembler-generated listing files to set break-points. The listings contain addresses relative to the beginning of the module, not the final, absolute addresses in the bound (linked) program.

You can also set a break-point range based only on a "starting point" and a "distance", without having to compute the start and end points manually:



Edit address range

You can specify ranges in the same way throughout the host software, no matter what you are doing:



Fill a block of Code memory

Chapter 3: Reference

Expression Input

Whenever and wherever the *iceMASTER* host software requests an address, count or length, you can enter an arbitrary expression (suitable for the particular context, of course). Expressions are composed of the following elements.

Identifiers

The first character in an identifier name cannot be a digit or the '$' character:

First character:	**_ ? @ a-z A-Z**
Subsequent characters:	**_ $ a-z A-Z 0-9**

Constants

Integer Constants

iceMASTER supports C-style constants as well as a variety of other popular formats. The following notations are used in subsequent definitions. A plural suffix (*s*) means "one or more of" the specified characters or digits. A suffix of *-opt* means that the item is optional (i.e., "zero or more of").

hex-digit:	**0-9 a-f A-F**
dec-digit:	**0-9**
non-zero-dec-digit:	**1-9**
oct-digit:	**0-7**
bin-digit:	**0-1**

Explicit Radix

The following constants contain an explicit radix (base) designator, either as a prefix or a suffix. Such constants will always be interpreted in the designated base, regardless of the current setting of the *Configure|Options|Expressions|Radix* command (see page 3-15). Following each definition are some examples of that particular style of constant.

0x *hex-digits*
0X *hex-digits*

Hexadecimal constant with explicit prefix (C style).

An optional suffix of '**u**' or '**U**' specifies an unsigned integer. This is generally unnecessary, as the host software treats all constants as unsigned.

An optional suffix of '**l**' or '**L**' specifies a long (32-bit) integer rather than a 16-bit integer.

The suffixes '**u**'/'**U**' and '**l**'/'**L**' can be used at the same time, in either order.

```
0x12
0xa
0x12751
```

X' *hex-digits*

Hexadecimal constant with explicit prefix (NSC style).

```
x'ab
x'45
x'123456
```

$ *hex-digits*

Hexadecimal constant with explicit prefix (Motorola style).

```
$ab
$0045
$123456
```

dec-digit hex-digits-opt **h**
dec-digit hex-digits-opt **H**

Hexadecimal constant with explicit suffix (Intel, et. al., style).

```
1ah
0054h
12345678h
```

dec-digits **t**
dec-digits **T**
dec-digits **.**

Decimal constant with explicit suffix.

```
12t
045.
```

non-zero-dec-digit dec-digits-opt **u**
non-zero-dec-digit dec-digits-opt **U**
non-zero-dec-digit dec-digits-opt **l**
non-zero-dec-digit dec-digits-opt **L**

Decimal constant with explicit "suffix" (C style).

A suffix of '**u**' or '**U**' specifies an unsigned integer. This is generally unnecessary, as the host software treats all constants as unsigned.

A suffix of '**l**' or '**L**' specifies a long (32-bit) integer rather than a 16-bit integer.

The suffixes '**u**'/'**U**' and '**l**'/'**L**' can be used at the same time, in either order.

```
5u
231
47ul
```

oct-digits **o**
oct-digits **O**
oct-digits **q**
oct-digits **Q**

Octal constant with explicit suffix.

```
12o
045o
```

0 *oct-digits*

Octal constant with explicit prefix (leading zero, C style).

An optional suffix of '**u**' or '**U**' specifies an unsigned integer. This is generally unnecessary, as the host software treats all constants as unsigned.

An optional suffix of '**l**' or '**L**' specifies a long (32-bit) integer rather than a 16-bit integer.

The suffixes '**u**'/'**U**' and '**l**'/'**L**' can be used at the same time, in either order.

```
012
055u
```

bin-digits **y**
bin-digits **Y**

Binary constant with explicit suffix.

```
1011y
011101Y
```

In all cases, if you do not supply an '**l**' or '**L**' suffix (or if such a suffix is not allowed) and the constant cannot fit in 16 bits, it will automatically be treated as a long value (32 bits).

If you are unsure of how a particular constant is being interpreted by the host software, just use the *Display/Alter|Expression* command (page 3-25) to see how the constant or constant expression is being treated.

No Explicit Radix

If you enter a number such as 10 or 101, it will be interpreted in the current default radix (base) as specified by the *Configure|Options|Expressions|Radix* command (see page 3-15). However, in certain contexts where it makes sense, the host software will temporarily override the current default. For example, when you enter a source line number as

```
#11
```

the software temporarily changes the default radix to 10 (decimal) before translating the '11'. In other cases, when the software displays a Dialog Box to obtain input, a prompt somewhere in the box (usually on the bottom border) indicates if the default radix has been changed temporarily.

In any case, you can always provide an explicit radix designator in the constant to override any default interpretation. For example, '#0x23' designates source line number 35 decimal, as does '#35t'.

In some cases, the software can automatically determine the appropriate radix based only on the characters and digits present in the constant. For example, **1A** can only be the hexadecimal number 0x1A, even though no explicit prefix or suffix is present.

Floating-Point Constants

The software supports both single-precision and double-precision floating-point constants.

dec-digits . *dec-digits exponent-opt*
dec-digits exponent

The *exponent* has the following form:

exponent: *exp-char sign-opt dec-digits*

exp-char: **e** or **E**

sign: + or -

If there is no suffix, the constant is double-precision floating-point (**double**). An optional suffix of '**f**' or '**F**' specifies that the constant is single-precision floating-point (**float**). An optional suffix of '**l**' or '**L**' explicitly specifies explicitly that the constant is double-precision floating-point (**double**).

Unlike C, "**1.**" is not a valid floating-point constant. You must always supply at least one digit after the decimal point (e.g., "**1.0**").

```
1.2
1.2f
5.6e5
```

Character Constants

The value of a character constant is the numerical value of the character interpreted as an integer. Internally, character constants are treated just as if you had entered an integer constant.

'\ *1-to-3-oct-digits* '
'\x *1-to-3-hex-digits* '
'\ *escaped-char* '
' *char* '

The first form uses an escaped sequence of one to three octal digits.

The second form uses an escaped sequence of one to three hexadecimal digits.

The third form specifies one of the following specially recognized escape characters:

Source Form	Internal Value	Meaning
\a	0x07	ANSI alert (bell)
\b	0x08	ANSI backspace
\e	0x1B	ANSI ESC
\f	0x0C	ANSI Formfeed
\n	0x0A	ANSI New line
\r	0x0D	ANSI Carriage return
\t	0x09	ANSI Horizontal tab
\v	0x0B	ANSI Vertical tab

If the backslash precedes a character that does not appear in the table above, the value used is the value of that character itself. That is, '\w' is the same as 'w'.

The fourth form represents the standard, "simple" case.

```
'\077'
'\xAB'
'\n'
'H'
```

String Constants

" *chars-opt* "

The form of each *char* in *chars-opt* is the same as that allowed for character constants. That is, all forms of escape sequences are allowed internally.

Currently, string constants can only be used as fill values in the *Display/Alter|(memory)|Fill* commands.

Expressions

The operators and operands which are allowed in expressions are grouped below in decreasing precedence (priority) order. The highest precedence operators and expression elements are listed first. To simplify the description, the term *expr* means any expression at that precedence level or higher, and the term *expression* means any expression at all.

Primary

identifier

Denotes a variable, label, module, function or SFR (Special Function Register) name. In the absence of any qualification on *identifier*, the software searches for the definition in the following order:

- local symbols in the current module
- global symbols

In the following descriptions, we use more meaningful names for identifiers to denote what kind of identifier is required in a particular context:

- *module-name*
- *function-name*
- *SFR-name*

constant

An integer constant, floating-point constant or character constant.

string

A character string constant (literal).

(*expression*)

A parenthesized expression. Parentheses can be used to change or clarify the order of operator precedence.

 # integer-constant

Source line number *'integer-constant'* in the current module.

module-name : *# integer-constant*

Source line number *'integer-constant'* in the module *module-name*.

module-name : *identifier*

Local variable or function *identifier* in module *module-name*.

module-name : *function-name* : *variable-name*

The local variable *variable-name* in the function *function-name* in module *module-name*.

C : *integer-constant*
X : *integer-constant*
D : *integer-constant*
I : *integer-constant*
B : *integer-constant*

Denotes a memory-space qualified constant. In most cases where the host software requests an address, you do not need to specify explicitly what memory space to use when you enter a constant or constant expression. For example, when setting a Code Memory break-point, if you enter **0x23**, the host software assumes that you mean location 0x0023 in Code Memory. Similarly, if you use labels or variable names, the symbol/debug information supplied in the program load module file normally tells in which memory space that symbol belongs.

However, if you want to change the value in a generic pointer (see page 3-13), you must tell the host software the memory space in which the address exists (which memory space the address is relative to).

Designator	Memory Space	Applicability			
		MCS-51	COP8	68HC11	68HC05
C	Code	x	x	x	x
D	Directly-addressable Internal Data	x	x		
I	Indirectly-addressable Internal Data	x			
X	External Data	x			
B	Bit-addressable Internal Data	x			

You can use a memory-space qualified constant to designate an address in other contexts, too. For example, in a program with no symbolic debug information, you can add the Watch Expression **X:0x23** to the Watch Window. This will allow you to monitor location 0x0023 in External Data memory.

Examples:

X:0x23 designates location 0x0023 in External Data memory

D:45h designates location 0x45 in Directly-addressable Internal Data/RAM memory

Postfix

expr . *member-name*

Structure or union member *member-name* in the **struct/union** expression *expr*. The *expr* must have a **struct** or **union** type.

expr . *integer-constant*

(MCS-51 only) Designates a particular bit in a bit-addressable SFR. *integer-constant* must be between 0 and 7, inclusive.

expr -> *member-name*

Structure or union member *member-name* in the **struct/union** pointed at by pointer expression *expr*. The *expr* must have the type "pointer to **struct**" or "pointer to **union**".

expr [*expression*]

Subscript expression. The entity being subscripted, *expr*, must have type "array of ..." or "pointer to ...". The subscript expression, *expression*, must be of an integral type.

Unary

* *expr*

Indirection through pointer expression *expr*, which must be of a "pointer to ..." type.

- *expr*

Unary minus. The *expr* can be either integral or a floating-point constant.

+ *expr*

Unary plus. The *expr* can be either integral or a floating-point constant.

Cast

(*type-name*) *expr*

Casting is intended for use in conjunction with a memory-space qualified constant (an address). The meaning of such an expression is "treat the location *expr* as having data type *type-name*".

The available *type-name*s, which are reserved words, are:

type-name	C Data Type	Size	Memory Layout
bit	bit	1 bit	
char	char	1 byte	
uchar	unsigned char	1 byte	
short	short	2 bytes	MSB, LSB
ushort	unsigned short	2 bytes	MSB, LSB
int	int	2 bytes	MSB, LSB
uint	unsigned int	2 bytes	MSB, LSB
long	long	4 bytes	MSW, LSW (each MSB, LSB)
ulong	unsigned long	4 bytes	MSW, LSW (each MSB, LSB)
float	float	4 bytes	IEEE single-precision floating-point
double	double	8 bytes	IEEE double-precision floating-point

MSB: Most Significant Byte (8 bits)
LSB: Least Significant Byte (8 bits)
MSW: Most Significant Word (16 bits, 2 bytes)
LSW: Least Significant Word (16 bits, 2 bytes)

Note that no conversions are performed on the value itself. The cast merely changes the data type that the host software associates with the designated address expression.

Multiplicative

expr * *expr*

Multiplication.

expr / *expr*

Division.

expr % *expr*

Modulo division.

Additive

expr + *expr*

Addition.

expr - *expr*

Subtraction.

Display Formats

This section describes the display formatting conventions used by the host software.

Addresses

Addresses displayed in the following contexts

- Watch Window
- *Display/Alter|Expression* command
- *Source/Symbols|Address* command
- *Source/Symbols|Alpha* command
- *Source/Symbols|Global* command
- *Source/Symbols|Local* command
- Pop-up Change Box
- Pop-up Browse Window

are of the form:

m:hhhh

where *m* is a character denoting the memory space and *hhhh* is the hexadecimal offset into that memory space. The possible values of *m* are:

Designator *m*	Memory Space	Applicability			
		MCS-51	COP8	68HC11	68HC05
C	Code	x	x	x	x
D	Directly-addressable Internal Data	x	x		
I	Indirectly-addressable Internal Data	x			
X	External Data	x			
B	Bit-addressable Internal Data	x			
N	(none / any / "number")	x	x	x	x

Type Names

Fundamental Types

Displayed Type Name	C Data Type	Size (Memory Layout)
(unknown)	-	*Configure*|*Options*|*Expressions*|*Unknown data type size* (see page 3-16)
(untyped)	-	
bit	**bit**	1 bit
char	**signed char**	1 byte
uchar	**unsigned char**	1 byte
short	**signed short**	2 bytes (MSB, LSB)
ushort	**unsigned short**	2 bytes (MSB, LSB)
int	**signed int**	2 bytes (MSB, LSB)
uint	**unsigned int**	2 bytes (MSB, LSB)
long	**signed long**	4 bytes (MSW (MSB, LSB), LSW (MSB, LSB))
ulong	**unsigned long**	4 bytes (MSW (MSB, LSB), LSW (MSB, LSB))
float	**float**	4 bytes (IEEE format)
double	**double**	8 bytes (IEEE format)
void	**void**	-
(label)	(statement label)	-

Bit Field Types

The type name displayed for structure members which are C bit fields shows the underlying integral type preceded by the bit offset into an object of that type and the number of bits in the field. The general format is:

offset:length_*type-name*

where 'offset' is the offset to the least significant bit in the field, 'length' is the field length (in bits) and *type-name* is the name of the underlying type. The offset 'offset' is zero-relative, starting at the least significant bit. For example, the bits in a **char** are numbered 7,...,0, with 7 being the most significant bit and 0 being the least significant bit.

See page 3-18 for an example of the display of a structure containing bit field members.

Derived/Aggregate Types

The general display formats are as follows, where "..." represents any data type name:

Display Format	Meaning
->...	"pointer to ..."
[n]...	"array of n ..."
()...	"function returning ..."
struct *tag-name*	(**struct**ure with tag name *tag-name*)
struct	(**struct**ure with no tag name)
union *tag-name*	(**union** with tag name *tag-name*)
union	(**union** with no tag name)

Unlike C, where data types read "from the inside out, obeying the parentheses", the data type names displayed by the host software read strictly from left-to-right:

Displayed Type Name	Example C Declaration	Data Type
->int	**int*** pi;	"pointer to **int**"
[5]int	**int** x[5]	"array of 5 **int**s"
()int	**int** f();	"function returning **int**"
[15]->int	**int*** y[15];	"array of 15 pointers to **int**"
->()->[10]->char	**char*** (*(*abc)())[10];	"pointer to a function returning a pointer to an array of 10 pointers to **char**"

Values

The display format of a value depends on that value's data type.

Fundamental Types

You can display values of the fundamental types in one of three formats:

a) Character (enclosed in apostrophes (')')

b) Decimal (signed or unsigned)

c) Hexadecimal

You control the display format for a particular data type with commands in the *Configure|Options|Expressions* pull-down menu (see page 3-20).

Pointers

The display format of a pointer value depends on the "nature" of the pointer.

Generic Pointers

Generic pointers are needed due to the Harvard architecture of the MCS-51. A generic pointer occupies three bytes in memory. The first byte contains a selector value/code denoting which memory space is being referenced (e.g., Code memory, External Data memory, etc.). The next two bytes contain the offset (0x0000-0xFFFF) into that memory space.

A generic pointer is used when the compiler does not know (at compile-time) what memory space a pointer variable may reference.

The general display format for a generic pointer is:

m:hhhh

where *m* is a character denoting the memory space and *hhhh* is the hexadecimal offset into that memory space. The possible memory space characters (*m*) are a subset of those listed in the table on page 3-10.

The display format for invalid (illegal, erroneous) generic pointers is:

?:*hhhh* **{***sshhhh***}**

An invalid generic pointer value is one with an illegal memory space selector byte value. This can occur for several reasons:

1) The application program has not yet initialized (assigned to) the pointer.
2) The application program has erroneously "clobbered" (written to) the memory space selector byte value in the pointer.
3) The linker has assigned (static) variables to memory locations so that local variables in functions which cannot be active at the same time share the same memory for their local variables. For example, if functions 'a' and 'b' can never be active at the same time, the linker may assign the local variables in function 'a' such that they use the same memory locations as the local variables in function 'b'. If function 'b' is currently active and you display a local pointer variable defined in function 'a', the value in that pointer variable may appear to be "bad".

Memory-Specific Pointers

Memory-specific pointers are used when the compiler "knows" at compile-time what memory space is being referenced by a particular pointer. Memory-specific pointers occupy only the number of bytes required to hold an address spanning a particular memory space. Memory-specific pointers do not contain a memory space selector byte.

The general display formats for memory-specific pointers are:

(*m*)*:hh*

(*m*)*:hhhh*

where *m*, *hh* and *hhhh* are the same as for the generic pointers. The memory space designator *m* is included to let you know which memory space is being referenced by the offset value *hh* or *hhhh*. It is enclosed in parentheses to let you know that the memory space is implicit (i.e., there is no memory space selector byte stored in memory).

NULL Pointers

The display format for a NULL pointer value is "**(null)**". The rules for detecting a NULL pointer value depend on the compiler you are using. See page 3-29 for specific details about NULL pointer values.

Structures, Unions and Arrays

The single-line "value" displayed for a structure, union or array object is the data type name itself ("struct...", "union..." or "[n]...").

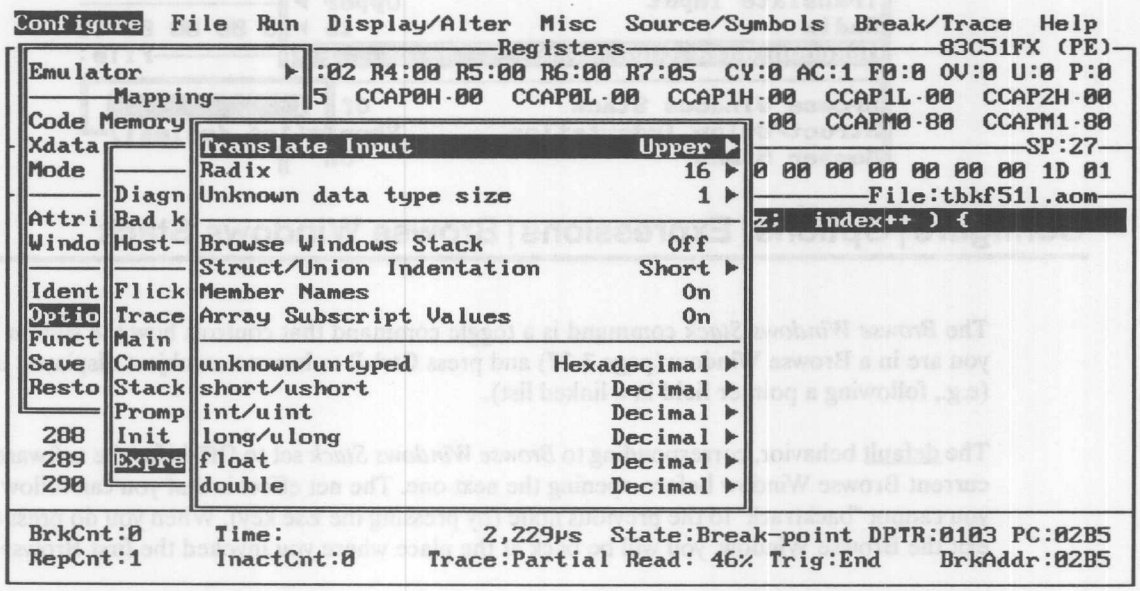
In the main-screen Watch Window and the pop-up Browse Windows, structures, unions and arrays are fully-expanded to show the individual members or elements. In these contexts, the individual members in a **struct/union** and the individual elements in an array immediately follow the single-line header for the structure, union or array. Each of these lines show the address, data type and value of the **struct/union** member or array element.

Functions

The value displayed for an entity of type "function returning..." is the name of the function itself. This may seem redundant and silly in the simple case, but if you are following (indirecting through) a pointer to a function, the software displays the name of the function pointed to by the pointer.

Configure | Options | Expressions Command

The *Configure | Options | Expressions* command displays a pull-down menu of commands to allow you to control expression input (symbols and numbers), numeric value formatting, data structure formatting and data structure browsing features.

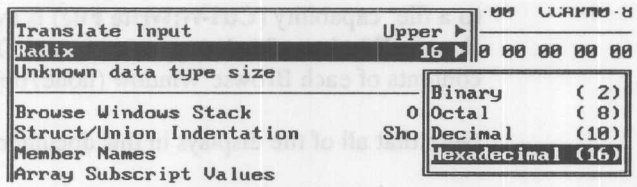


Case conversion to be applied to keyboard input and symbolic input from files

Configure | Options | Expressions | Radix

The *Radix* command allows you to specify the default radix (base) for numbers you enter that contain no explicit radix prefix or suffix. If you enter a number such as 10 or 101, it will be interpreted in the current default radix (base) as specified by this command. However, in certain contexts where it makes sense, the host software will temporarily override the current default. For example, when you enter a source line number as

#11



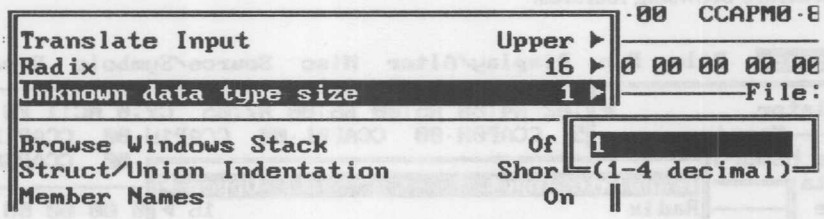
the software temporarily changes the default radix to 10 (decimal) before translating the '11'. In other cases, when the software displays a Dialog Box to obtain input, a prompt somewhere in the box (usually on the bottom border) indicates if the default radix has been changed temporarily.

In any case, you can always provide an explicit radix designator in the constant to override any default interpretation. For example, '#0x23' designates source line number 35 decimal, as does '#35t'.

In some cases, the software can automatically determine the appropriate radix based only on the characters and digits present in the constant. For example, 1A can only be the hexadecimal number 0x1A, even though no explicit prefix or suffix is present.

Configure | Options | Expressions | Unknown data type size

The *Unknown data type size* command tells the host software how many bytes to read, write or display for a value/symbol whose data type is "(unknown)" or "(untyped)".



Configure | Options | Expressions | Browse Windows Stack

The *Browse Windows Stack* command is a toggle command that controls how the system behaves when you are in a Browse Window (page 3-27) and press **Ctrl-B** to browse an object displayed in that window (e.g., following a pointer field in a linked list).

The default behavior, corresponding to *Browse Windows Stack* set to **Off**, is for the software to "close" the current Browse Window before opening the next one. The net effect is that you can follow a pointer, but you cannot "backtrack" to the previous node (by pressing the **Esc** key). When you do press the **Esc** key to exit the Browse Window, you will be back at the place where you invoked the first Browse Window.

A *Browse Windows Stack* setting of **On** causes successive pop-ups of a Browse Window to stack one on top of the other (recursively). You can backtrack to any of the previous Browse Windows by pressing the **Esc** key one or more times. As nice as it is, the reason that this is not the default system behavior is that you could cause the host software to overflow its own runtime stack by traversing too deeply into list or tree or whatever.

However, even when you have *Browse Windows Stack* set to **Off** (default setting), there is a way to at least maintain a record of the nodes/objects you have visited while browsing. The "write the data in this window to a file" capability (**Ctrl-W:Write File**) is available in a Browse Window, regardless of the setting of the *Browse Windows Stack* toggle command. Thus, at each step along the way, you can write the entire contents of each Browse Window (node/object), successively appending to the same file.

Note that all of the displays in this document were generated with *Browse Windows Stack* set to **On**.

The *Struct/Union Indentation* command controls the indentation of structure and union members which are themselves structures, unions or arrays. The two indentation options available are '*Short*' and '*Long*'. The default is '*Short*' indentation.

With '*Short*' indentation, the members/elements of embedded structures, unions or arrays are indented two spaces from the beginning of the name of the "parent" member:

```
X:0164 ->struct Srt_Nd newnode
 Addr   Data Type        Value
X:0104 struct Srt_Nd   struct Srt_Nd
X:0104 union             lf_link     union
X:0104 ->struct Srt_N      srx_left X:0124
X:0104 ->struct Srt_N      srx_fore X:0124
X:0107 union             rb_link     union
X:0107 ->struct Srt_N      srx_right X:0144
X:0107 ->struct Srt_N      srx_back  X:0144
X:010A [4]char           srt_value   [4]char
X:010A char                [0] 0x41 'A'   65
X:010B char                [1] 0x42 'B'   66
X:010C char                [2] 0x43 'C'   67
X:010D char                [3] 0x00 '.'    0
X:010E uchar             srt_vtype   0x04 '.'   4
X:010F char              srt_char    0x4D 'M'   77
X:0110 int               srt_int     25
X:0112 uint              srt_flags   20
X:0114 long              srt_long    25
X:0118 ulong             srt_ulong   25
X:011C float             srt_float   12345.678
```

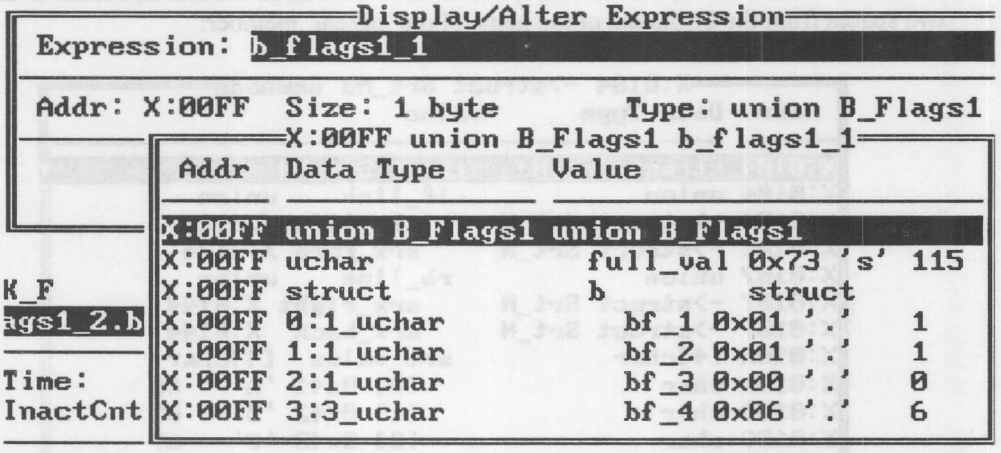
With '*Long*' indentation, the members/elements of embedded structures, unions or arrays are indented two spaces from the beginning of the value field of the "parent" member:

```
X:0164 ->struct Srt_Nd newnode
 Addr   Data Type        Value
X:0104 struct Srt_Nd   struct Srt_Nd
X:0104 union             lf_link     union
X:0104 ->struct Srt_N                  srx_left X:0124
X:0104 ->struct Srt_N                  srx_fore X:0124
X:0107 union             rb_link     union
X:0107 ->struct Srt_N                  srx_right X:0144
X:0107 ->struct Srt_N                  srx_back  X:0144
X:010A [4]char           srt_value   [4]char
X:010A char                            [0] 0x41 'A'   65
X:010B char                            [1] 0x42 'B'   66
X:010C char                            [2] 0x43 'C'   67
X:010D char                            [3] 0x00 '.'    0
X:010E uchar             srt_vtype   0x04 '.'   4
X:010F char              srt_char    0x4D 'M'   77
X:0110 int               srt_int     25
X:0112 uint              srt_flags   20
X:0114 long              srt_long    25
X:0118 ulong             srt_ulong   25
X:011C float             srt_float   12345.678
```

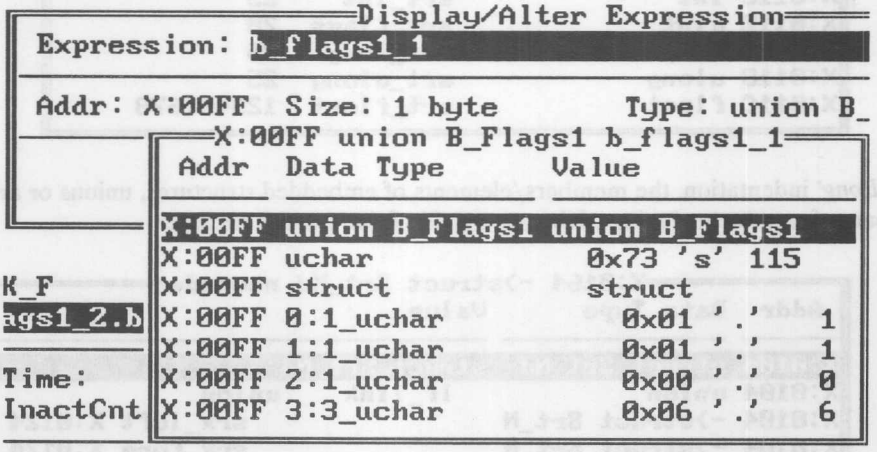
Note that this command has a visible effect only if the setting of the *Configure | Options | Expressions | Member Names* toggle command is '**On**' (see page 3-18).

The *Member Names* toggle command controls whether or not the host software will display structure and union member names when displaying the individual members of a structure or union. The default is **'On'**.

This is an example of a union display generated with the *Member Names* toggle set to **'On'** (the default):



Here is what the same display looks like with the *Member Names* toggle set to **'Off'**:



The *Array Subscript Values* toggle command controls whether or not the host software will display an array element's subscript value (enclosed in square brackets) before the actual value of the array element itself. The default is '**On**'.

This is an example of an array display generated with the *Array Subscript Values* toggle set to '**On**' (the default):

X:00FF [5]int scores

Addr	Data Type	Value
X:00FF	[5]int	[5]int
X:00FF	int	[0] 0
X:0101	int	[1] 1
X:0103	int	[2] 4
X:0105	int	[3] 9
X:0107	int	[4] 16

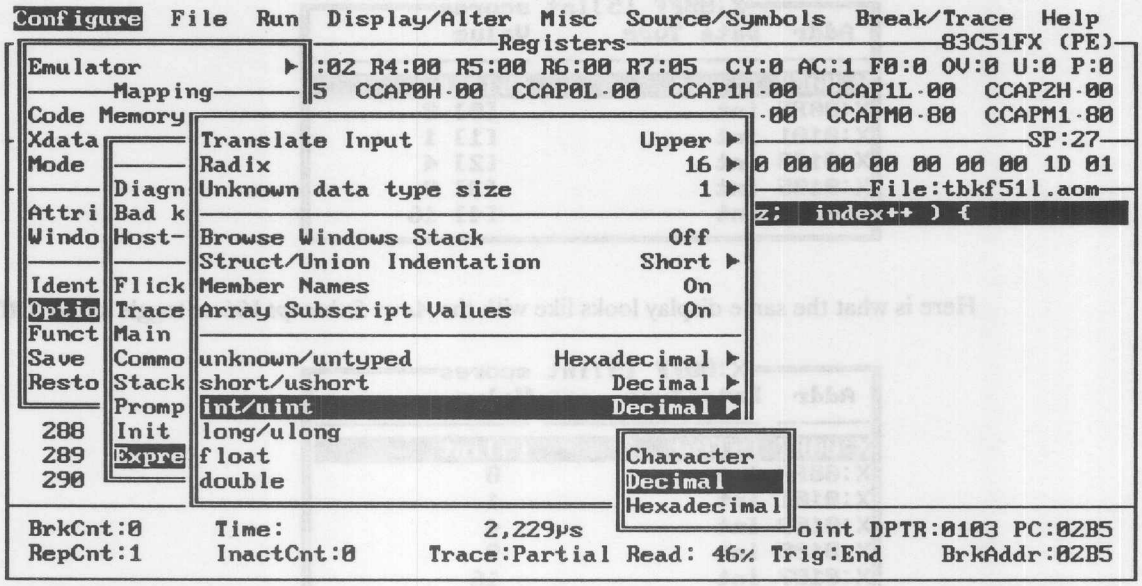
Here is what the same display looks like with the *Array Subscript Values* toggle set to '**Off**':

X:00FF [5]int scores

Addr	Data Type	Value
X:00FF	[5]int	[5]int
X:00FF	int	0
X:0101	int	1
X:0103	int	4
X:0105	int	9
X:0107	int	16

Configure | Options | Expressions | unknown/untyped
Configure | Options | Expressions | short/ushort
Configure | Options | Expressions | int/uint
Configure | Options | Expressions | long/ulong
Configure | Options | Expressions | float
Configure | Options | Expressions | double

These commands control the display format to be used for values of the fundamental (basic) data types.



Display in decimal format

The available choices are:

Command	Controls Display Formatting for These Data Types	Display Formatting Options Available
Configure | Options | Expressions | unknown/untyped	(unknown) & (untyped)	Character, Decimal (signed or unsigned), or Hexadecimal
Configure | Options | Expressions | short/ushort	**short** & **unsigned short**	
Configure | Options | Expressions | int/uint	**int** & **unsigned int**	
Configure | Options | Expressions | long/ulong	**long** & **unsigned long**	
Configure | Options | Expressions | float	**float**	
Configure | Options | Expressions | double	**double**	

Note that there is no control needed for the **char** and **unsigned char** types, since values of these 1-byte types are always displayed in all three possible formats: hexadecimal, character and signed/unsigned decimal (in that order).

If you select Decimal format, signed or unsigned decimal notation will be used depending on the actual data type of the value (e.g., signed decimal for **int**, unsigned decimal for **unsigned int**).

The *Configure | Options | Expressions* commands operate, for the most part, independently of one another.

For example, the following structure display was generated with everything set to the "maximum" (*Struct/Union Indentation* = **Long**, *Member Names* = **On**, *Array Subscript Values* = **On**):

X:0164 ->struct Srt_Nd newnode

Addr	Data Type	Value	
X:0104	struct Srt_Nd	struct Srt_Nd	
X:0104	union	lf_link	union
X:0104	->struct Srt_N		srx_left X:0124
X:0104	->struct Srt_N		srx_fore X:0124
X:0107	union	rb_link	union
X:0107	->struct Srt_N		srx_right X:0144
X:0107	->struct Srt_N		srx_back X:0144
X:010A	[4]char	srt_value	[4]char
X:010A	char		[0] 0x41 'A' 65
X:010B	char		[1] 0x42 'B' 66
X:010C	char		[2] 0x43 'C' 67
X:010D	char		[3] 0x00 '.' 0
X:010E	uchar	srt_vtype	0x04 '.' 4
X:010F	char	srt_char	0x4D 'M' 77
X:0110	int	srt_int	25
X:0112	uint	srt_flags	20
X:0114	long	srt_long	25
X:0118	ulong	srt_ulong	25
X:011C	float	srt_float	12345.678

Here is the same structure display generated with everything set to the "minimum" (*Struct/Union Indentation* = **Short**, *Member Names* = **Off**, *Array Subscript Values* = **Off**):

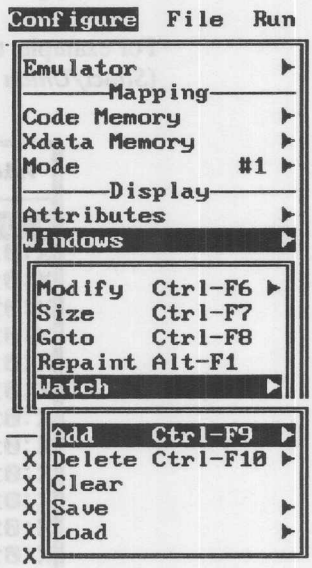
X:0164 ->struct Srt_Nd newnode

Addr	Data Type	Value
X:0104	struct Srt_Nd	struct Srt_Nd
X:0104	union	union
X:0104	->struct Srt_N	X:0124
X:0104	->struct Srt_N	X:0124
X:0107	union	union
X:0107	->struct Srt_N	X:0144
X:0107	->struct Srt_N	X:0144
X:010A	[4]char	[4]char
X:010A	char	0x41 'A' 65
X:010B	char	0x42 'B' 66
X:010C	char	0x43 'C' 67
X:010D	char	0x00 '.' 0
X:010E	uchar	0x04 '.' 4
X:010F	char	0x4D 'M' 77
X:0110	int	25
X:0112	uint	20
X:0114	long	25
X:0118	ulong	25
X:011C	float	12345.678

Watch Window

Commands

The *Configure|Windows|Watch* pull-down menu contains five commands to allow you to define and manipulate the content of the Watch Window.



The **Add** command lets you add any non-constant expression to the Watch Window. Note that the host software reevaluates the address designated by the watch expression each time the Watch Window is updated. Thus, if you add the expression *my_array[i]* to the Watch Window, the host software recomputes the subscript value of *i* each time the software updates or repaints the Watch Window.

The **Delete** command deletes (removes) a single, designated expression from the Watch Window.

The **Clear** command deletes (removes) all watch expressions from the Watch Window.

The **Save** command saves (writes) all the watch expressions in the Watch Window to a file of your choice. You can later load (add) these saved watch expressions to the Watch Window by selecting the *Configure|Windows|Watch|Load* command below.

The **Load** command loads (restores) into the Watch Window the watch expressions saved in a file by the *Configure|Windows|Watch|Save* command above.

The various commands, or the entire menu, may be inactive (inaccessible) depending on the current state of the system. For example, if the Watch Window is empty, the Delete, Clear and Save commands will be inactive (inaccessible).

Display Format

The Watch Window has four columns, showing the address of the watch expression, the data type of the watch expression, the watch expression itself and the current value of the watch expression:

```
Configure  File  Run  Display/Alter  Misc  Source/Symbols  Break/Trace  Help
Module:TA                          Source                        File:takf511.aom
  76            aj = (ai * 2) + main_loop_factor;
  77            ary[ai] = aj;
                                   Watch
X:008E uint             ai                      6
X:00BA uchar            ary[ai]                 0x2F '/'  47
X:0084 [10]uchar        ary                     [10]uchar
X:0084 uchar            ..................        [ 0] 0x24 '$'  36
X:0085 uchar            ..................        [ 1] 0x26 '&'  38
X:0086 uchar            ..................        [ 2] 0x28 '('  40
X:0087 uchar            ..................        [ 3] 0x2A '*'  42
X:0088 uchar            ..................        [ 4] 0x2C ','  44
X:0089 uchar            ..................        [ 5] 0x2E '.'  46
X:008A uchar            ..................        [ 6] 0x2F '/'  47
X:008B uchar            ..................        [ 7] 0x31 '1'  49
X:008C uchar            ..................        [ 8] 0x33 '3'  51
X:008D uchar            ..................        [ 9] 0x35 '5'  53
X:00A8 ->struct List    p_list                  (null)
C:0000?struct List      *p_list                 struct List
C:0000?uint             ..................        list_uival 517
C:0002?->struct List    ..................        list_p_nxt ?:FFFF{CEFFFF}
X:0080 float            iteration               38.5
Begin emulation
```

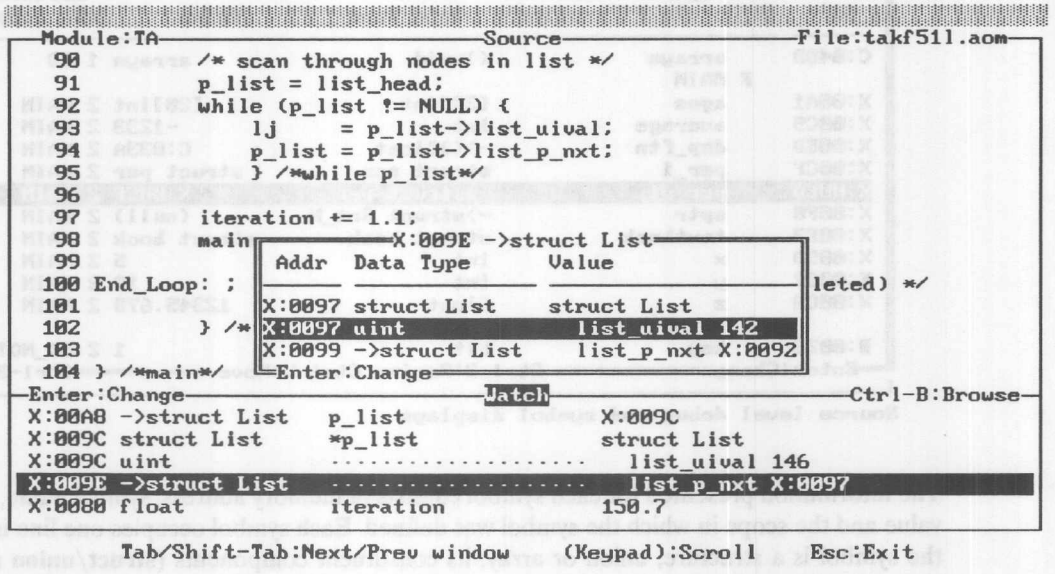
If the watch expression is a structure, union or array type, then there will also be a line for each constituent component in the watch expression: each member in a **struct/union** or each element in an array. In this case, the watch expression field contains ".....".

Every time the host software updates or repaints the Watch Window, it highlights the value field of each watch expression or constituent component if the current value is different from the previous value. This highlighting is done to notify you of the change in value. Similarly, if the address associated with the watch expression or constituent component changes, the software highlights the address field. (The address associated with a watch expression can change if you are watching an expression such as *my_array[i]*, where the actual array element being monitored changes as the value of *i* changes.)

If the host software encounters an "illegal" value in evaluating the address of a watch expression, it flags the watched object with a question mark (?) between the address column and the data type column. This can happen if the watch expression involves indirection through a pointer and the value of that pointer is NULL or illegal (see page 3-13).

Browsing

You can enter Change Mode to browse the Watch Window by pressing the **Tab** key (which is simply a convenient shortcut for the *Configure|Windows|Goto* command) one or more times until the Watch Window title is highlighted:



As you scroll up or down in the Watch Window, the software will display prompts on the top border of the Watch Window indicating what actions you can perform on the highlighted item:

Enter:Change – (upper-left border) The currently highlighted item is a fundamental value or a pointer, either of which you can change. Pressing the **Enter** key pops up a Change Box (see page 3-26), allowing you to change the item's current value.

Ctrl-B:Browse – (upper-right border) The currently highlighted item is a pointer which can be "followed". Pressing the **Ctrl-B** key sequence pops up a Browse Window (see page 3-27), allowing you to browse/inspect the object pointed to by the pointer value.

The following *Source/Symbols* windows display the internal symbol table sorted in various ways:

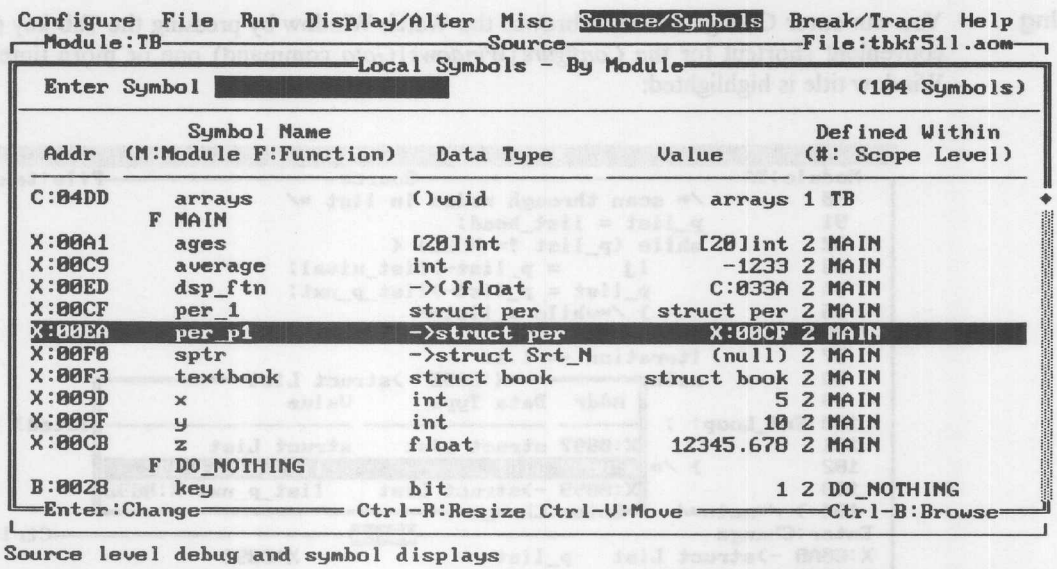
Source/Symbols|Global: global (PUBLIC) symbols, sorted alphabetically

Source/Symbols|Local: local symbols, sorted alphabetically within each scope

Source/Symbols|Alpha: global and local symbols, sorted alphabetically

Source/Symbols|Address: global and local symbols, sorted by address

All these windows have the same basic display format. Following is a display generated by the *Source/Symbols|Local* command:



Source level debug and symbol displays

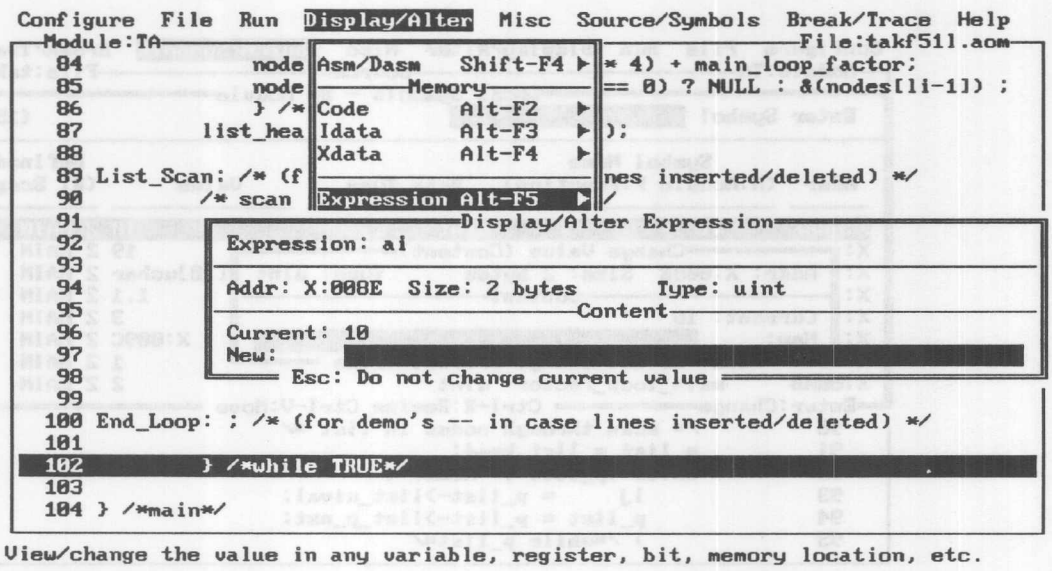
The information presented for each symbol consists of memory address, symbol name, data type, current value and the scope in which the symbol was defined. Each symbol occupies one line in these displays. If the symbol is a structure, union or array, its constituent components (**struct/union** members or array elements) are not shown in these displays. To see the individual **struct/union** members or array elements, you must press the "browse" key combination (**Ctrl-B:Browse** – see below).

As you scroll up or down in these *Source/Symbols* windows, the software will display prompts on the bottom border of the window indicating what actions you can perform on the highlighted symbol:

Enter:Change – (bottom-left border) The currently highlighted symbol's data type is one of the fundamental types or it is a pointer. Pressing the **Enter** key pops up a Change Box (see page 3-26), allowing you to change the symbol's current value.

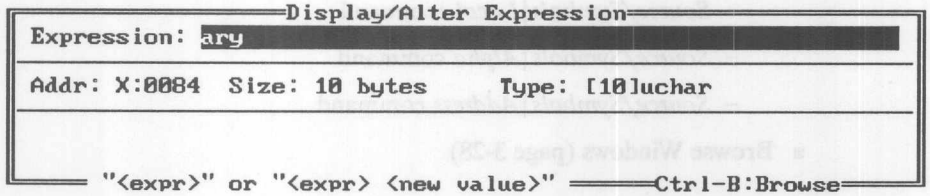
Ctrl-B:Browse – (bottom-right border) The currently highlighted symbol is an array, structure or union, or it is a pointer (which can be "followed"). Pressing the **Ctrl-B** key sequence pops up a Browse Window (see page 3-27), allowing you to browse/inspect the array elements, **struct/union** members or the object pointed to by the pointer variable.

The *Display/Alter|Expression* command pops up a dialog box allowing you to view or change the value stored in a location designated by an arbitrary expression.



The dialog box shows the address, size (in bytes or bits) and data type of the expression you enter. If the expression designates a location whose content can be changed, the "Current:" field shows the value currently stored in that location.

If the expression you enter designates a structure, union or array value, the software shows the address, size and data type of the expression. Additionally, the **Ctrl-B:Browse** prompt appears on the bottom-right border of the box to indicate that you can pop up a Browse Window (page 3-27) to inspect and/or change the individual members in the **struct/union** or the individual elements in the array:

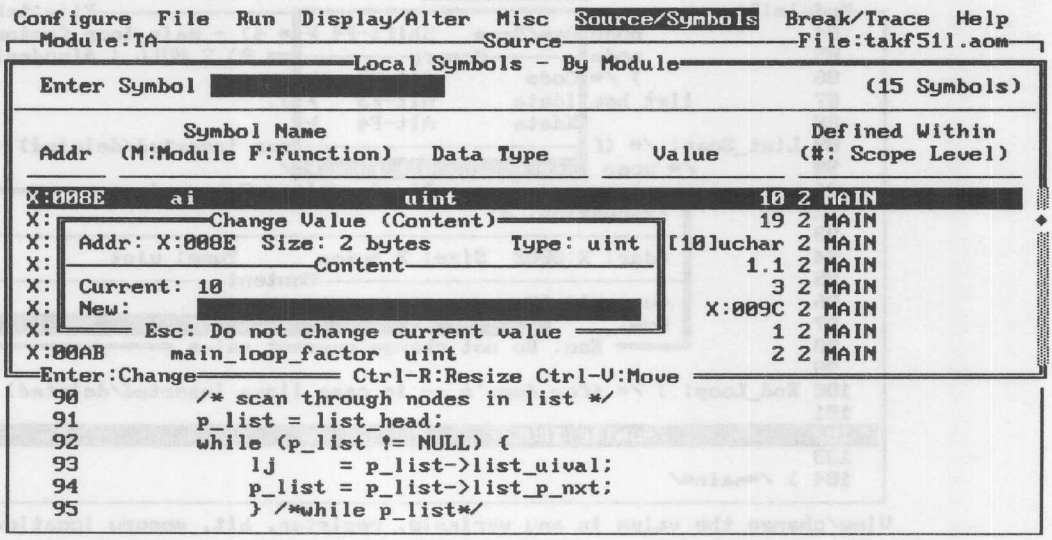


If the expression you enter designates a "pointer to..." value, you have the option of either changing the pointer, by entering a new value, or "following" the pointer, by pressing **Ctrl-B** (Browse).

Note: Currently, the expression entered in the "New:" field is evaluated for address, not for value. For example, if you enter *i*, the address of the variable *i* is used as the new value, not the content of the variable *i*.

Change Box

A Change Box is very much like the *Display/Alter|Expression* command's pop-up dialog box. The difference is that, with a Change Box, the expression or variable whose value is to be changed is already known.



Source level debug and symbol displays

A Change Box pops up when you press the **Enter** key in response to the **Enter:Change** prompt in the following contexts:

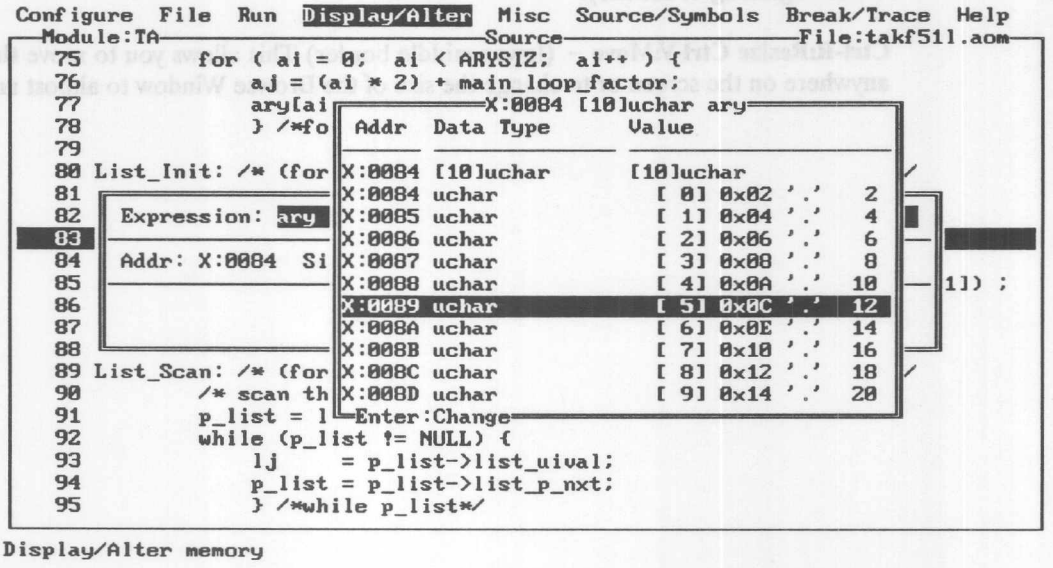
- Watch Window (page 3-23)
- Source/Symbols windows (page 3-24)
  - *Source/Symbols|Global* command
  - *Source/Symbols|Local* command
  - *Source/Symbols|Alpha* command
  - *Source/Symbols|Address* command
- Browse Windows (page 3-28)

Browse Windows

A Browse Window pops up to allow you to view (inspect) and/or change the

- members in a structure or union
- elements in an array
- object designated by a pointer value (i.e., to "follow" the pointer)

Here is a Browse Window popped-up from the *Display/Alter|Expression* command to examine the individual elements in the array *ary*:



A Browse Window pops up when you press the **Ctrl-B** key in response to the **Ctrl-B:Browse** prompt in the following contexts:

- Watch Window (page 3-23)
- Source/Symbols windows (page 3-24)
  - *Source/Symbols|Global* command
  - *Source/Symbols|Local* command
  - *Source/Symbols|Alpha* command
  - *Source/Symbols|Address* command
- *Display/Alter|Expression* command (page 3-25)
- Browse Windows (e.g., to follow a pointer which is a **struct/union** member or an array element)

Structures, unions and arrays are fully "expanded" in a Browse Window to show all constituent components: each member in a **struct/union** or each element in an array. This process applies iteratively if a component is itself a structure, union or array.

The *Configure|Options|Expressions|Browse Window Stack* toggle command (page 3-16) controls how the system behaves when you are in a Browse Window and press **Ctrl-B** to browse an object displayed in that window (e.g., following a pointer field in a linked list).

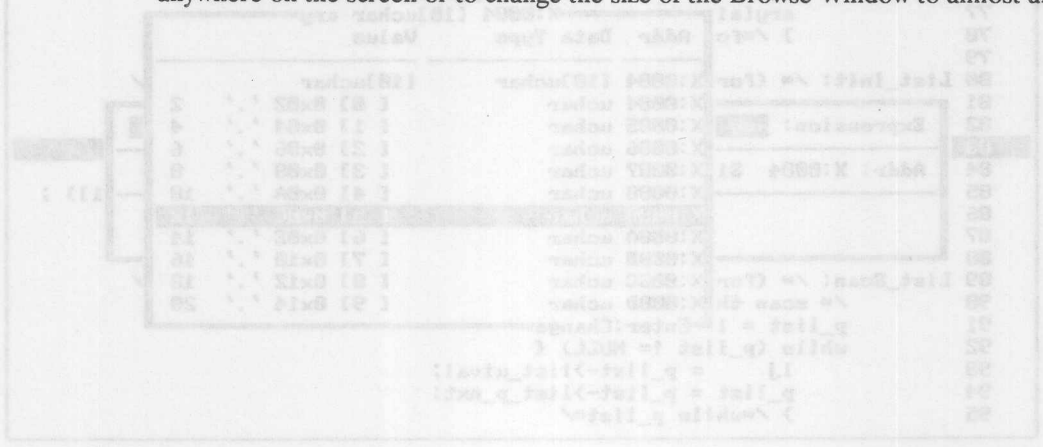
The following keys are available in a Browse Window:

Enter:Change – (lower-left border) The currently highlighted item is a fundamental value or a pointer, either of which you can change. Pressing the **Enter** key pops up a Change Box (see page 3-26), allowing you to change the item's current value.

Ctrl-B:Browse – (lower-right border) The currently highlighted item is a pointer which can be "followed". Pressing the **Ctrl-B** key sequence pops up another Browse Window, allowing you to browse/inspect the object pointed to by the pointer value.

Ctrl-W:Write – (upper-right border) Pressing **Ctrl-W** allows you to write the entire contents of the Browse Window to a file of your choice. The software will prompt you for the name of the file and ask you whether you want to append to that file (add to the end of the file) or overwrite it (start over at the beginning of the file).

Ctrl-R:Resize Ctrl-V:Move – (lower-middle border) This allows you to move the Browse Window anywhere on the screen or to change the size of the Browse Window to almost any size you want.



Compilers

This section describes how to get the most out of the capabilities in the *iceMASTER* host software in combination with the particular compiler product you are using. *iceMASTER* currently provides full data type support (structures, unions, arrays, etc.) for the compilers listed in this section. More comprehensive support for other compilers will be added in the near future.

Keil/Franklin C51

This information applies to version V3.07 of C51 (C compiler) and version V2.7 of L51 (linker). It should be accurate for later versions of these products and may even be valid for earlier versions.

Recommended Compilation Options

DEBUG	Include debug information in the object file.
OBJECTEXTEND	Include full data type definition debug information in the object file.
NOREGPARMS	Do not pass function parameters in registers.
OPTIMIZE(3)	Do not put auto variables in registers.

Recommended Linking Options

DEBUGLINES	Include source line number debug information in the output file.
DEBUGPUBLICS	Include public (global) symbol information in the output file.
DEBUGSYMBOLS	Include all symbol information in the output file.

Pointers

The Keil/Franklin C51 compiler implements both generic pointers and memory-specific pointers (see page 3-13).

NULL

The value of a NULL generic pointer is a pointer value with all three bytes containing zero (even the selector byte). This is true regardless of whether the pointer is being read (e.g., for comparison to NULL) or written (i.e., NULL being assigned to the pointer).

'typedef' Names

The *iceMASTER* host software is internally set up to display **typedef** names, where appropriate. However, the compiler does not output any information about **typedefs**. The compiler always outputs the underlying type. For example, in

```
typedef  struct abc {...}  ABC;
ABC      xyz;
```

There is no information about 'ABC' in the AOM file (Absolute Object Module file, load module file). The data type of 'xyz' is recorded as '**struct** abc', not '**typedef** ABC'. Likewise, for

```
typedef  unsigned char  logical;
logical  bool_val;
```

there is no information about 'logical' and the type of 'bool_val' is recorded as **unsigned char**.

Enumeration Types

The *iceMASTER* host software is internally set up to display **enum**eration values as they were declared in the C source program (e.g., 'RED', 'GREEN', 'YELLOW', rather than 0, 1, 2). However, the compiler does not output **enum**eration type information. The AOMF-specified type of **enum** variables always seems to be **signed int**.

'double' Types

The data type **double** (double-precision floating-point) is implemented by the compiler as **float** (single-precision floating-point). The data type information in the AOM file for variables declared as 'double' reflects this.

Batch "Make" Files

The batch "make" files TAKF51.BAT and TBKF51.BAT show how the example programs TA.C and TB.C were compiled and linked.

This information applies to version V4.23E/DXT of C-51 (C compiler) and version V4.44D/DXT of XLINK (linker). It should be accurate for later versions of these products and may even be valid for earlier versions.

Recommended Compilation Options

-r0n Put only symbolic information (no source code) into the object file. Do not add extra NOPs at source line boundaries.

Do not specify '-r1...' or '-r2...': The *iceMASTER* emulators break emulation before the instruction at a break-point is executed. Thus, no extra NOPs are needed in the generated code.

Warning: Do not specify '-r0i...' (add #include file source code to object file) and do not forget to specify the 'n' in '-r0n'. The host software ignores the source images embedded in the load module file. *iceMASTER* locates your original source file on disk and reads that file when displaying source images. Additionally, the line numbers recorded in the resultant load module file (for '-r0i...') will not match the original source file found on your disk.

Recommended Linking Options

-Fdebug Create the load module file in the IAR/Archimedes DEBUG format (.DBG).

Pointers

The IAR/Archimedes C-51 compiler implements only generic pointers; there are no memory-specific pointers (see page 3-13).

NULL

The value of a NULL generic pointer is dependent on the context.

Reading In comparing a pointer value to NULL, only the offset portion (*hhhh*) is examined (for both bytes being zero). The value in the memory space selector byte is ignored.

Writing When NULL (zero) is assigned to a pointer, all three bytes in the pointer (including the memory space selector byte) are set to zero.

'typedef' Names

The *iceMASTER* host software is internally set up to display **typedef** names, where appropriate. However, the compiler does not output any information about **typedefs**. The compiler always outputs the underlying type. For example, in

```
typedef  struct abc {...}  ABC;
ABC      xyz;
```

There is no information about 'ABC' in the .DBG file (load module file). The data type of 'xyz' is recorded as '**struct** abc', not '**typedef** ABC'. Likewise, for

```
typedef  unsigned char  logical;
logical  bool_val;
```

there is no information about 'logical' and the type of 'bool_val' is recorded as **unsigned char**.

Enumeration Types

The *iceMASTER* host software is internally set up to display **enum**eration values as they were declared in the C source program (e.g., 'RED', 'GREEN', 'YELLOW', rather than 0, 1, 2). However, the compiler does not output **enum**eration type information. The .DBG-specified type of **enum** variables always seems to be **signed int**.

'double' Types

The data type **double** (double-precision floating-point) is implemented by the compiler as **float** (single-precision floating-point). The data type information in the .DBG file for variables declared as 'double' reflects this.

Batch "Make" Files

The batch "make" files TAIA51.BAT and TBIA51.BAT show how the example programs TA.C and TB.C were compiled and linked.

Appendix A: Source Window & Assembler/Disassembler

Source Window

The **Source Window** is used to display the source program and/or disassembled instructions in Code Memory. The current module (see the *Source/Symbols | Current Module* command) is displayed on the left side of the top border and the filename of the currently loaded program (see the *File | Load* command) is displayed on the right side of the top border.

The information in this window may be displayed in 'Code', 'Mixed' or 'HLL' mode. 'Code' means just an assembly language disassembly of the code will be displayed. 'Mixed' means that HLL source lines (e.g., C language), if available, will be displayed along with the disassembled code. 'HLL' means that just the HLL source lines will be displayed. The default is 'HLL' but can be changed under the *Options* column of the *Configure | Windows | Modify* command.

The starting address of the code displayed is at the PC address and cannot be changed. Note that during single step (*Run | Step*) or slow motion operation (*Run | Slow Motion*) the starting address of the screen does not change until the flow of execution moves out of the code that appears in the window, at which time a new window full of code will be displayed. This is intended to keep the Dynamically Annotated Code visible as long as possible.

In Change Mode, the Source Window also allows you to control the emulation environment.

> You can enter Change Mode to browse/peruse the Source Window by pressing the **Tab** key (which is simply a convenient shortcut for the *Configure | Windows | Goto* command) one or more times until the Source Window title is highlighted.

Once you are in Change Mode, by pressing ←, → or **Enter**, the *Source Window Control* Menu becomes available.

```
──────────Source Window Control──
Browse  Change_Module  Mode  Label_Synch  Set_Break  Assemble
──Mode=HLL──────────────────────Label-Synch=On──
```

Source Window Control

The *Source Window Control* Menu is a menu bar window which allows you to browse through or change your code, change the display mode, turn disassembly label synchronization ON and OFF, set or clear simple break-points and set temporary break-points. The commands are:

Browse

The *Browse* command allows you to quickly reposition the highlight bar in the Source Window. If the Source Window display mode is *Code* or *Mixed*, you will be prompted for an address to go to. If the Source Window display mode is *HLL* (source only), you will be prompted for a line number. However, you may enter any address expression (numerically or symbolically) or line number at either prompt. Note that if the display mode is *HLL* and you want to switch to a different module, use the *Change Module* command.

Change Module

In *HLL* (source-only) or *Mixed* display mode, you can use the *Change Module* command to display (switch to) a different module in the Source Window.

Mode

The *Mode* command calls a Pull-down Menu from which you may set the display mode for the Source Window. The available choices are *Code*, *Mixed* and *HLL* (source only).

Mode | Code

The *Code* command changes the display mode of the Source Window to a disassembled view of code memory.

Mode | Mixed

The *Mixed* command changes the display mode of the Source Window to a disassembled view of code memory, interspersed with HLL source line images for code-generating source lines.

Mode | HLL

The *HLL* command changes the display mode of the Source Window to a source-only view of the HLL source module corresponding to the currently highlighted line. This display mode shows both code-generating and non-code-generating source lines.

Label-Synch

The *Label-Synch* command is used to toggle disassembly label synchronization ON and OFF in the 'Code' and 'Mixed' display modes. When Label-Synch is ON, the Host Software uses existing code labels to verify that disassembly addresses are on instruction boundaries. If they are not, addresses will be adjusted so that the disassembly does begin on an instruction boundary. If Label-Synch is OFF, no such checks are made and the disassembly will start at the specified address.

Set Break

The *Set Break* command calls a Pull-down Menu from which you may set or clear simple break-points and set a temporary break-point.

Set Break | Toggle

The *Toggle* command toggles (on or off) a permanent simple break-point at the currently highlighted line. Permanent simple break-point locations are flagged with the ▷ character and may be modified in the *Break/Trace | Set* menu.

Note that **Ctrl-B** (toggle break-point) performs the same function as the *Set Break | Toggle* command.

Set Break | Run Until

The *Run Until* command sets a temporary break-point at the currently highlighted line and then begins emulation immediately from the current PC. This command is essentially the same as the primary *Run | Until* command, except that the temporary break-point address is implicitly denoted as the currently highlighted line in the Source Window.

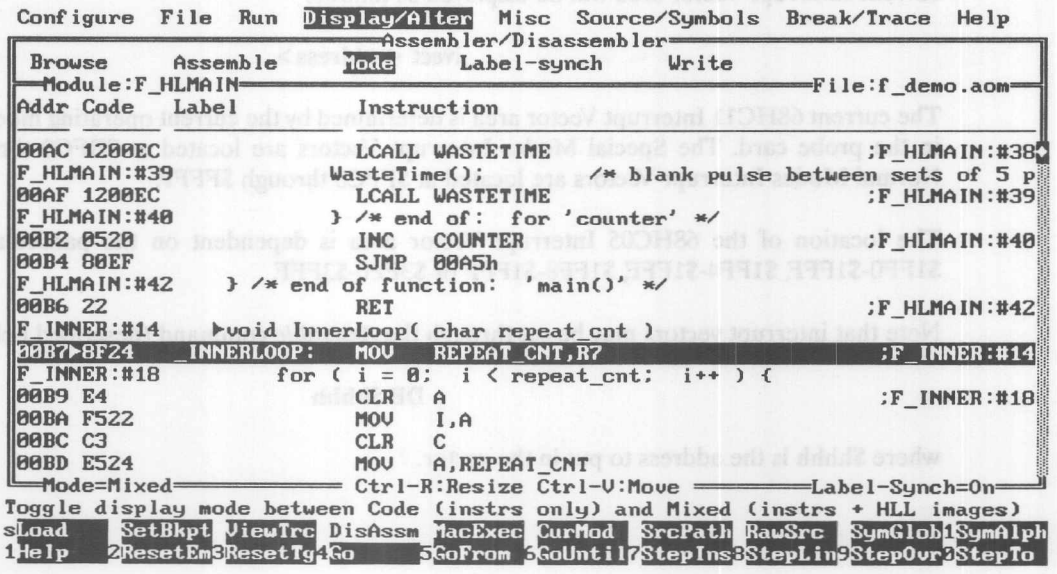
Note that this "set temporary break-point and start emulation" function may also be accomplished directly from the source window by pressing **Ctrl-T** (set temporary break-point and go).

Assemble

The *Assemble* command invokes the *Assembler/Disassembler* menu (*Display/Alter|Asm/Dasm* command) to allow you to browse through or change (patch) code.

Display/Alter | Asm/Dasm

The *Display/Alter|Asm/Dasm* command calls Assembler/Disassembler Menu Bar Window from which you may assemble instructions into code memory and browse through the disassembled code memory. Browse mode is active on entrance to the *Asm/Dasm* Menu.



Note that the right side of the top border line (under the Menu Bar) shows the filename of any loaded code file, the left side of the bottom border line shows the Display Mode, the center of the bottom border line shows current key information and the right side of the bottom border line shows the Label-synch mode. The commands are:

Display/Alter | Asm/Dasm | Browse

The *Browse* command calls a dialog box from which you may enter an address expression. If an address is entered the display will be repainted starting at the specified address.

Note that if an absolute address is entered and the Label-synch Mode is OFF, the address may or may not correspond to an instruction boundary and may result in an 'out-of-synch' disassembly.

Note for iceMASTER-68HC11 and iceMASTER-68HC05 disassemblies: Several opcode mnemonics have aliases (alternative mnemonic names). In such cases, the alternative name will be shown as a comment next to the disassembled instruction:

68HC11		68HC05	
Mnemonic	Alias	Mnemonic	Alias
ASL	LSL	BCC	BHS
ASLA	LSLA	BCS	BLO
ASLB	LSLB	DECX	DEX
ASLD	LSLD	INCX	INX
BCC	BHS	LSL	ASL
BCS	BLO	LSLA	ASLA
		LSLX	ASLX

You can use either form shown above when patching in new instructions using the single-line assembler (*Display/Alter|Asm/Dasm|Assemble* command).

In addition, for the iceMASTER-68HC11 and iceMASTER-68HC05, disassembled code memory at the current Interrupt Vector area will be displayed as follows:

ivect <address>

The current 68HC11 Interrupt Vector area is determined by the current operating mode of the processor in the probe card. The Special Modes Interrupt Vectors are located at $BFC0 through $BFFF. The Normal Modes Interrupt Vectors are located at $FFC0 through $FFFF.

The location of the 68HC05 Interrupt Vector area is dependent on the particular chip derivative: $1FF0-$1FFF, $1FF4-$1FFF, $1FF8-$1FFF or $3FF0-$3FFF.

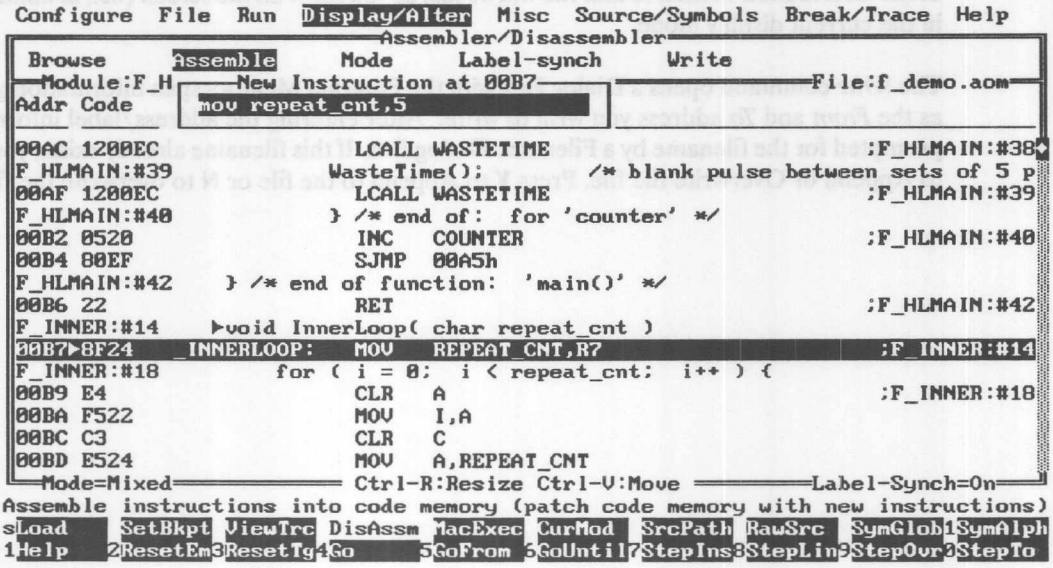
Note that interrupt vectors may be set through the *Assemble* command (described below) using

DB $hhhh

where $hhhh is the address to put in the vector.

Display/Alter | Asm/Dasm | Assemble

The *Assemble* command turns on Assemble Mode. In this mode you may enter new instructions (and labels) into code memory a single line at a time, at the current (highlighted) address. The address at which to add the instruction may be changed by using *Browse* mode (described above) or by using the ↓ or ↑ keys to increment or decrement the address.



When entering an instruction, symbolic information may be used. This includes the ability to define a new label at the current address and to use symbolic names in the operand part of the instruction. When a new label is entered, it is inserted into the internal symbol table as a global (public) symbol.

Note that if the assembled instruction does not contain the same number of bytes as the original instruction at the current address you will be notified and asked to verify the operation through a Confirmation Box. Note that the *Assemble* command is enabled only for a line of machine code, *NOT* for HLL source statements.

To exit Assemble Mode press **Esc** to return to the Menu Bar.

Display/Alter | Asm/Dasm | Mode

The *Mode* command is used to toggle the display mode between 'Code' Mode and 'Mixed' Mode. The 'Code' display Mode means that just assembly language instructions are displayed. The 'Mixed' display Mode (available only if HLL source is available) means that HLL source images are interspersed with the assembly code.

Display/Alter | Asm/Dasm | Label-synch

The *Label-synch* command is used to toggle disassembly label synchronization ON and OFF. When Label-synch is ON, the Host Software uses existing code labels to verify that disassembly addresses are on instruction boundaries. If they are not the Host Software will adjust the address so the disassembly begins on an instruction boundary. If the Label-synch mode is OFF, no such checks are made and the disassembly will start at the specified address.

Note that if Label-synch Mode is OFF, the specified address may or may not correspond to an instruction boundary and may result in an 'out-of-synch' disassembly.

Display/Alter | Asm/Dasm | Write

The *Write* command is used to write the disassembled contents of code memory to a disk file. The format of the information written to that file will be just as you see it on the screen (i.e., in human-readable form), in the current display mode.

The *Write* command opens a Dialog Box with the Program Memory span information given to you as well as the *From* and *To* address you wish to write. After entering the address/label information, you will be prompted for the filename by a Filename Dialog Box. If this filename already exists, you will be requested to Append or Overwrite the file. Press **Y** to Append to the file or **N** to overwrite the file.

Appendix B: Virtual Memory Support

Command Line Options

-vme <pages_per_allocation_request>
-vmf <num_bufs> <buf_size>
-vmfn <file_name>

These options control how virtual heap memory will be used. Enabling virtual heap memory allows the Host Software to use memory located outside the conventional 0-640Kb memory area normally usable by a DOS program. This capability is useful if you have a program with a large number of symbols which cannot all fit into the symbol table in conventional memory.

By default, the virtual heap memory is disabled and the Host Software builds the symbol table in conventional memory when loading a program file. If you ever see the following message at the top of the screen

```
"Not enough unused memory available to continue ... press any key."
```

the Host Software has used up all the conventional memory and, generally, cannot continue. If this happens, you should reinvoke the Host Software, specifying one or more of the command line options (-vme, -vmf and -vmfn) to enable virtual heap memory usage. However, note that loading files (building the internal symbol table) and some other commands within the Host Software (e.g., listing all symbols alphabetically) may be much slower when virtual heap memory is enabled.

There are two basic kinds of backing store media (real memory/storage) used to support the virtual heap memory:

1) LIM EMS 3.2/4.0 Expanded Memory, and

2) a file

The fastest medium is expanded memory and you should use this form if:

1) You have an expanded memory card in your computer (any type of PC). To allow access to this expanded memory, you must have the appropriate 'DEVICE=..' statement in your CONFIG.SYS file; see the documentation accompanying your expanded memory board for the details on how to do this, or

2) You have a '386 or '486 PC with no expanded memory, but with additional memory (extended memory) above the 1Mb boundary. In such a case, you can use the EMM386.EXE device driver supplied with DOS to simulate expanded memory in extended memory. A typical CONFIG.SYS file for this configuration would contain the following lines:

   DEVICE=C:\DOS\HIMEM.SYS
   DEVICE=C:\DOS\EMM386.EXE

   You can add additional parameters to the DEVICE command for EMM386.EXE to control the amount of expanded memory to be created (simulated) in your system.

If your computer does not fit either of the two categories above, but it does have extended memory (memory beyond the 1Mb boundary), you can use a file for virtual heap backing store. However, rather than using a file on your hard disk, create a RAM disk in extended memory and, using the '-vmfn' command line option, designate the backing store file as a file on that RAM disk drive. This is tremendously faster than using an actual hard disk file.

-vme <pages_per_allocation_request>

If you have Expanded Memory (LIM-compatible) in your computer, the Host Software can use it as a backing store medium for virtual heap memory. To/ enable access to Expanded Memory, you must have the appropriate 'DEVICE=...' statement in your 'CONFIG.SYS' file. See the documentation accompanying your Expanded Memory board for the specific details on how to do this.

The '-vme <pages_per_allocation_request>' option specifies the number of pages (16Kb per page) of Expanded Memory to request from the Expanded Memory Manager (EMM) each time the Host Software needs more Expanded Memory.

Whenever a program requests some number of logical pages of expanded memory from the EMM, those pages are internally associated with a 'handle'. The total number of available handles, as well as the total number of available logical pages of expanded memory, are resources which could easily be exhausted. If the total number of EMM handles in your system is small, make sure that the value of '-vme <pages_per_allocation_request>' is comparatively large. This will ensure that the available handles are not exhausted before the available logical pages of expanded memory.

To disable use of Expanded Memory as a backing store medium altogether, specify '-vme 0'.

-vmf <num_bufs> <buf_size>

Specifies the number and size of the real-memory buffers to be used to support file backing store as a medium for virtual memory. If either <num_bufs> or <buf_siz> is zero, file backing store will not be used at all (i.e., it will be disabled). The integer specified for <buf_size> is the number of K-bytes to allocate for each real-memory buffer.

You can change the path/name of the default backing store file ($VHEAP$) using the '-vmfn' option.

To disable use of the file backing store medium altogether, specify '-vmf 0'.

-vmfn <file_name>

If your system has only Extended Memory, but no Expanded Memory, you can significantly speed up access to the backing store file by creating a RAM disk in Extended Memory and then specifying the backing store file name as a file on that RAM disk (e.g., '-vmf E:\TMPFILE', where E: is the RAM disk drive).

Misc | Virtual Memory Command

The *Misc | Virtual Memory* command displays the following information about the current usage of virtual heap memory:

Expanded Memory Backing Store

EMM Version is the version number of the Expanded Memory Manager (EMM) device driver installed in your system. You can use EMM versions 3.2 and beyond.

Page Frame Segment is the segment address of the 64K area called the page frame. This will be somewhere above the conventional memory area used by DOS (0-640K) and below the 1M boundary of the upper memory area. The actual address depends on your computer's configuration.

Mappable Physical Pages is the number of physical pages in the upper memory area (640K-1M) of your computer into which expanded memory pages can be mapped (or swapped). The number here is the sum of the:

1) number of pages in the Page Frame (usually 4),

2) number of additional mappable pages above the 640K boundary.

Note that your system may have other mappable pages below the 640K boundary, but the Host Software will not use these pages.

This number represents the amount of expanded memory that the Host Software can reference directly at any particular time. The larger the value, the more likely that the memory area/node being referenced is directly accessible, and the less likely that the Host Software will need to swap pages to access the desired memory area/node.

Total Pages is the total number of expanded memory pages in your computer. Each page contains 16K of memory.

Pages Available is the number of pages currently available for allocation by the EMM to the Host Software. The remainder, if any, of the expanded memory pages in your system are unavailable for use by the Host Software because they are allocated to other programs or device drivers.

Pages Allocated to Host Software is the number of pages of expanded memory currently allocated to the Host Software.

Currently In Use by Host Software is the percentage of expanded memory which the Host Software is currently using from the pool of expanded memory pages allocated to the Host Software. When this percentage reaches 100%, the Host Software will request more expanded memory pages from the EMM; the '-vme' command line option (page B-2) specifies how many pages will be requested.

File Backing Store

Real-Memory Buffers is the number of conventional memory buffers being used to swap blocks in from and out to the backing store file. The '-vmf' command line option (page B-2) specifies how many buffers to use.

Buffer (Block) Size is the size of each conventional memory buffer used to swap blocks in from and out to the backing store file. The '-vmf' command line option (page B-2) specifies this buffer size.

The product of Real-Memory Buffers times Buffer (Block) Size represents the amount of real memory that the Host Software can reference directly at any particular time. The larger the value, the more likely that the memory area/node being referenced is directly accessible, and the less likely that the Host Software will need to swap blocks to/from the backing store file to access the desired memory area/node.

File Blocks Allocated is the number of blocks (of size Buffer (Block) Size each) currently needed by the Host Software. Essentially, this is the current size of the backing store file.

Currently In Use by Host Software is the percentage of file backing store which the Host Software is currently actually using. When this percentage reaches 100%, the Host Software will increase the size of the backing store file to create more virtual memory.